masai.

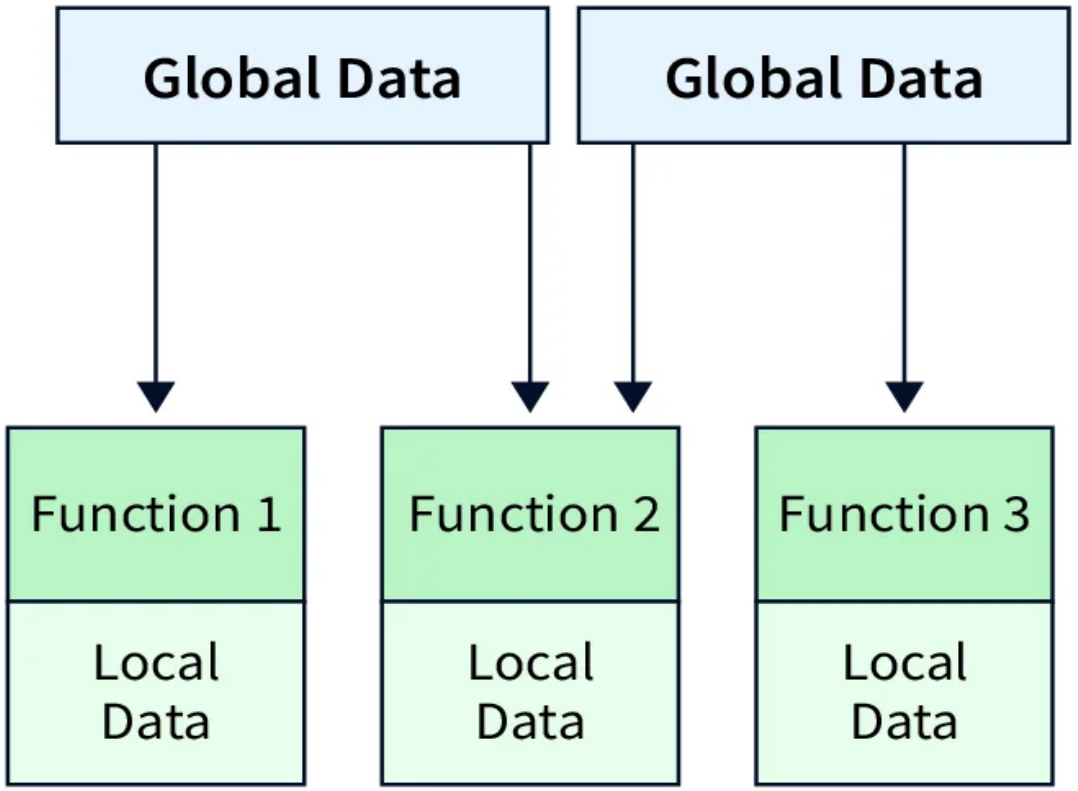
INTRODUCTION TO

# OBJECT ORIENTED PROGRAMMING

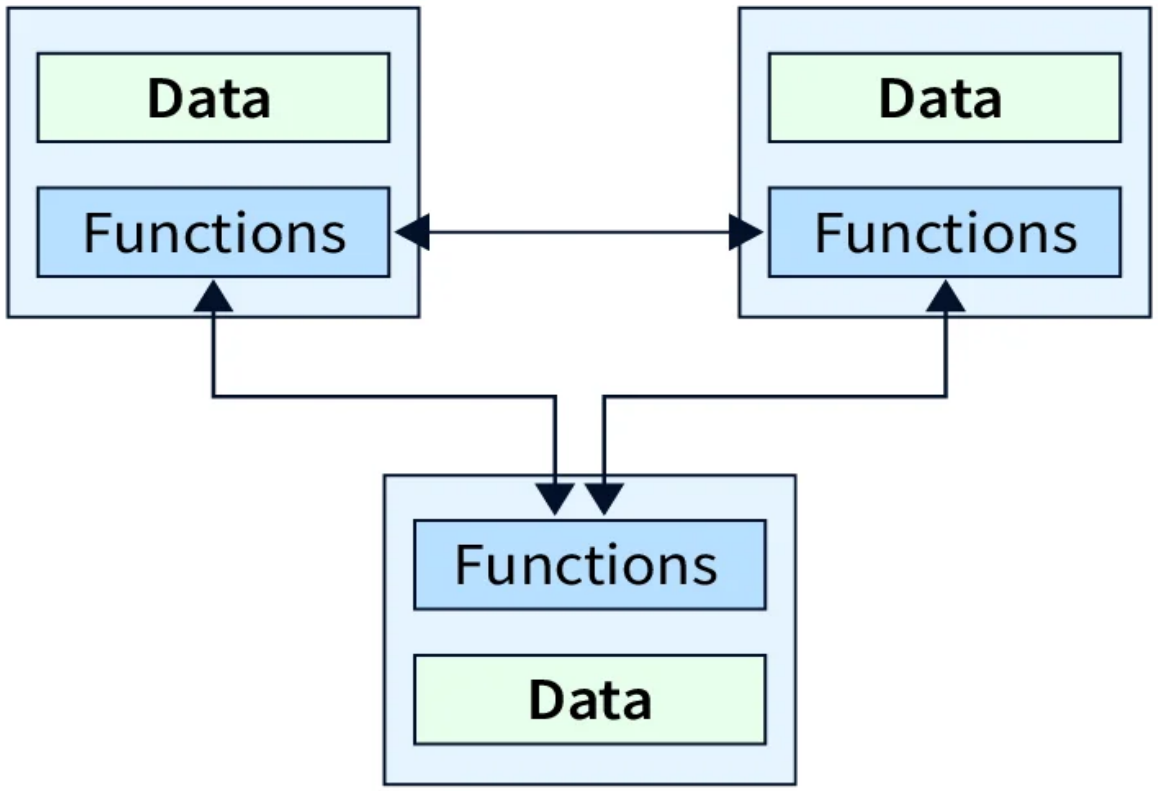

# What is object-oriented programming?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic.
An object can be defined as a data field that has unique attributes and behavior

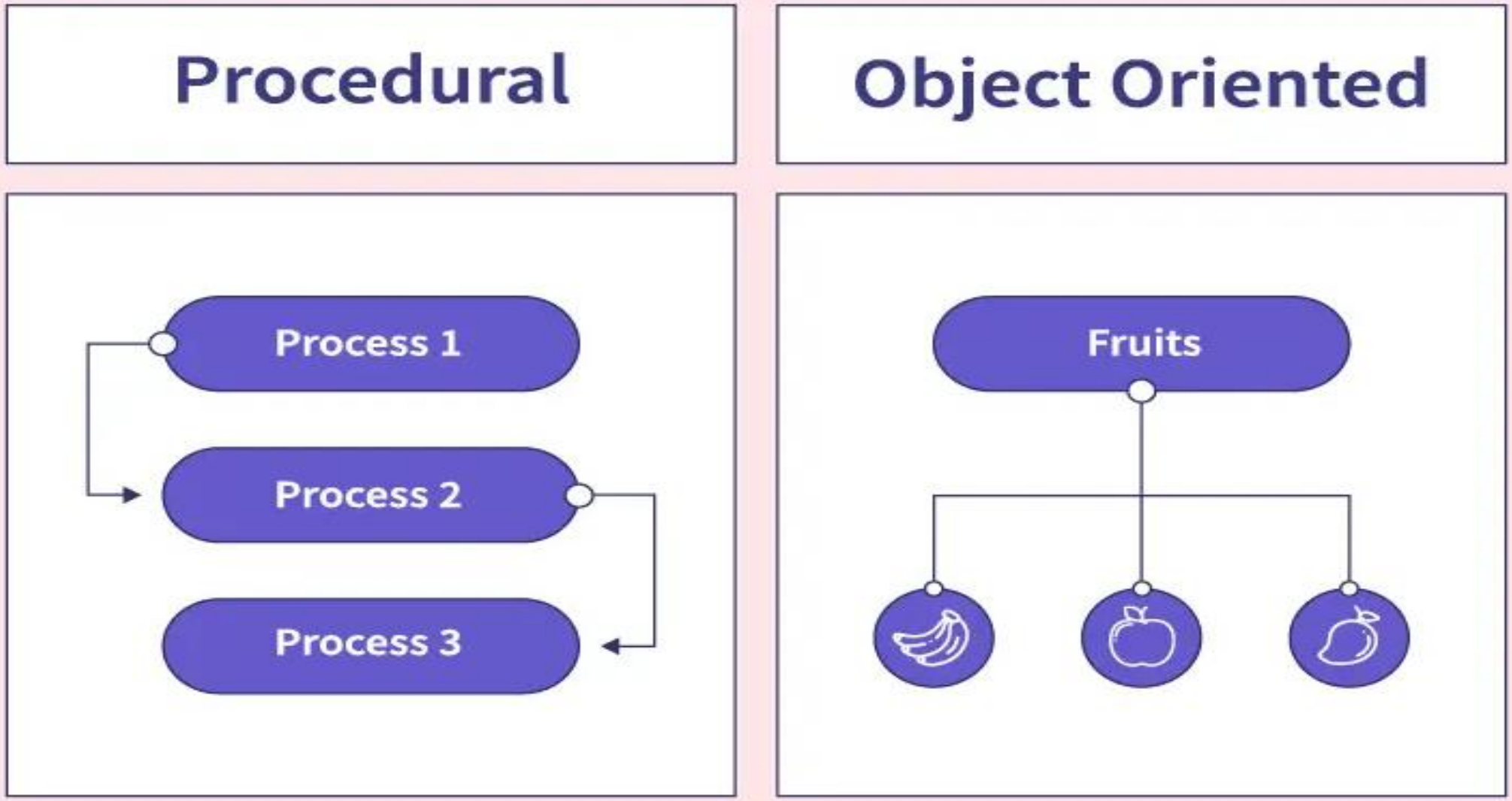
## Procedural Oriented Programming

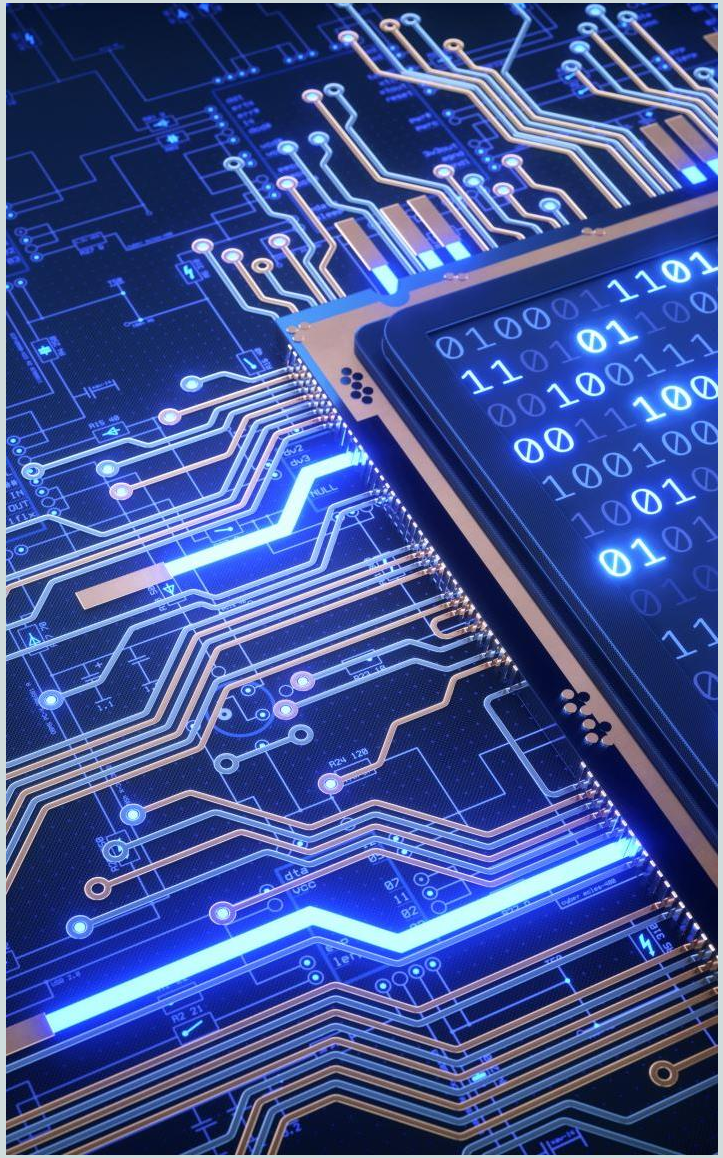



## Object Oriented Programming




SCALER Topics


Procedural
Object Oriented
Process 1
Process 2
Process 3
Fruits



**OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. This includes programs for manufacturing and design, as well as mobile applications; for example, OOP can be used for manufacturing system simulation software.**

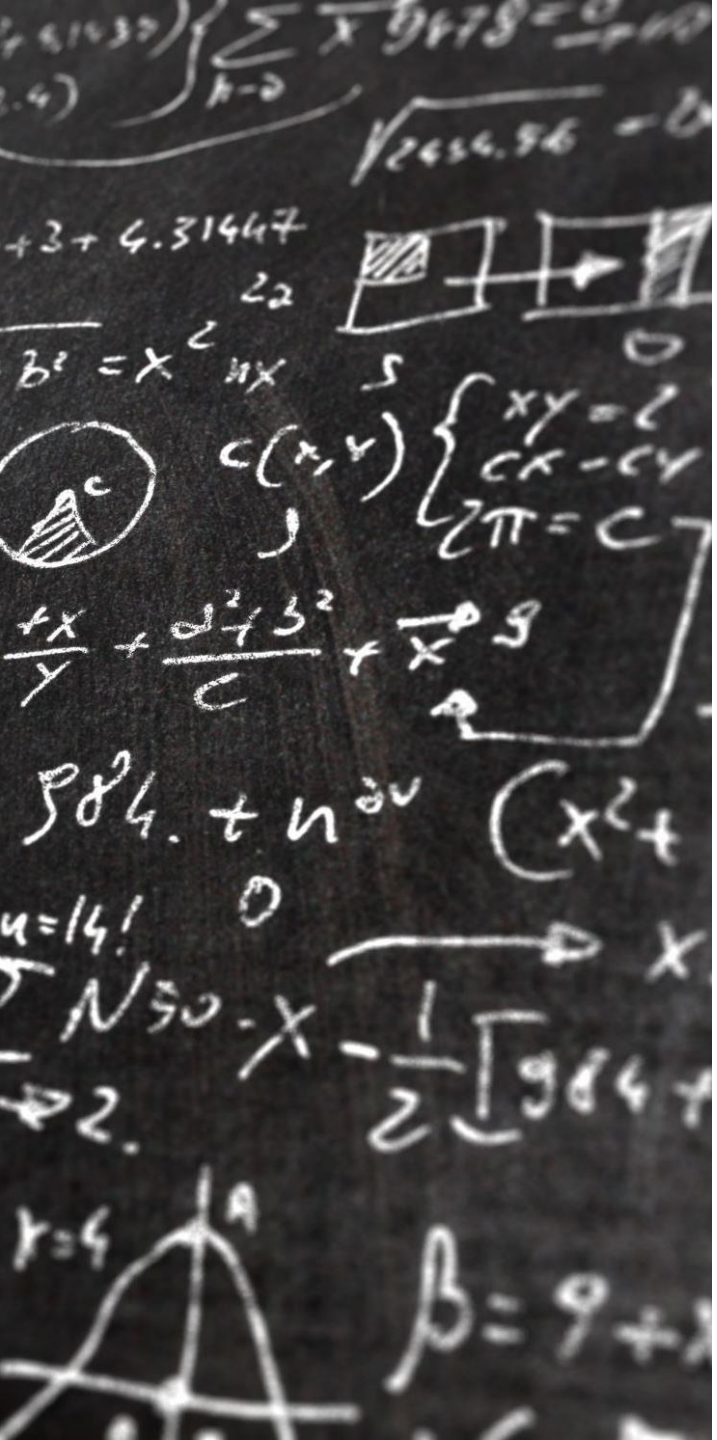
The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability and efficiency.

The first step in OOP is to collect all of the objects a programmer wants to manipulate and identify how they relate to each other -- an exercise known as

**data modeling.**

**Examples of an object can range from physical entities, such as a human being who is described by properties like name and address, to small computer programs, such as widgets.**

**Once an object is known, it is labeled with a class of objects that defines the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method. Objects can communicate with well-defined interfaces called messages.**



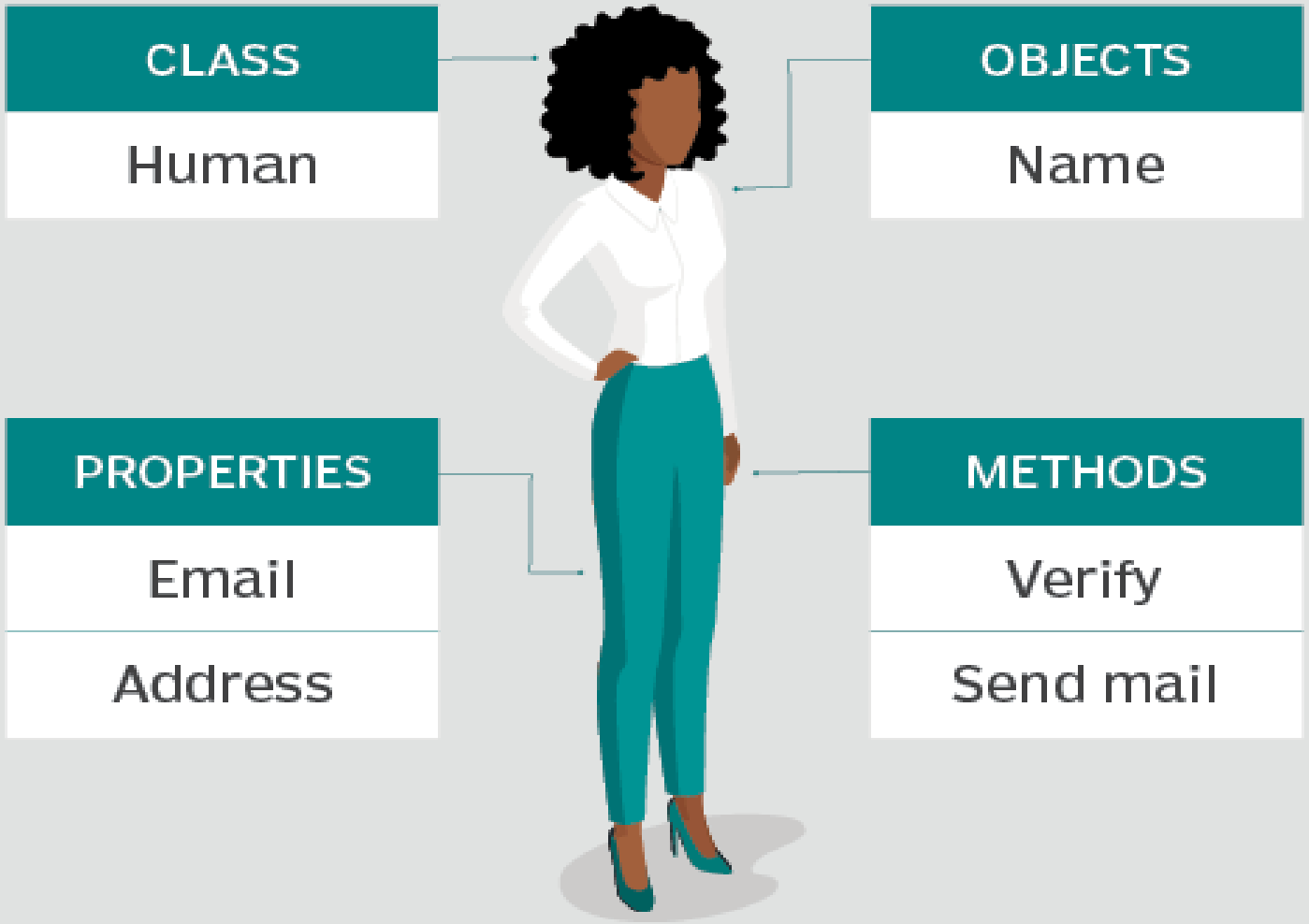
- **What is the structure of object-oriented programming?**
- **The structure, or building blocks, of object-oriented programming include the following:**

- **Classes are user-defined data types that act as the blueprint for individual objects, attributes and methods.**
- **Objects are instances of a class created with specifically defined data. Objects can correspond to real-world objects or an abstract entity. When class is defined initially, the description is the only object that is defined.**

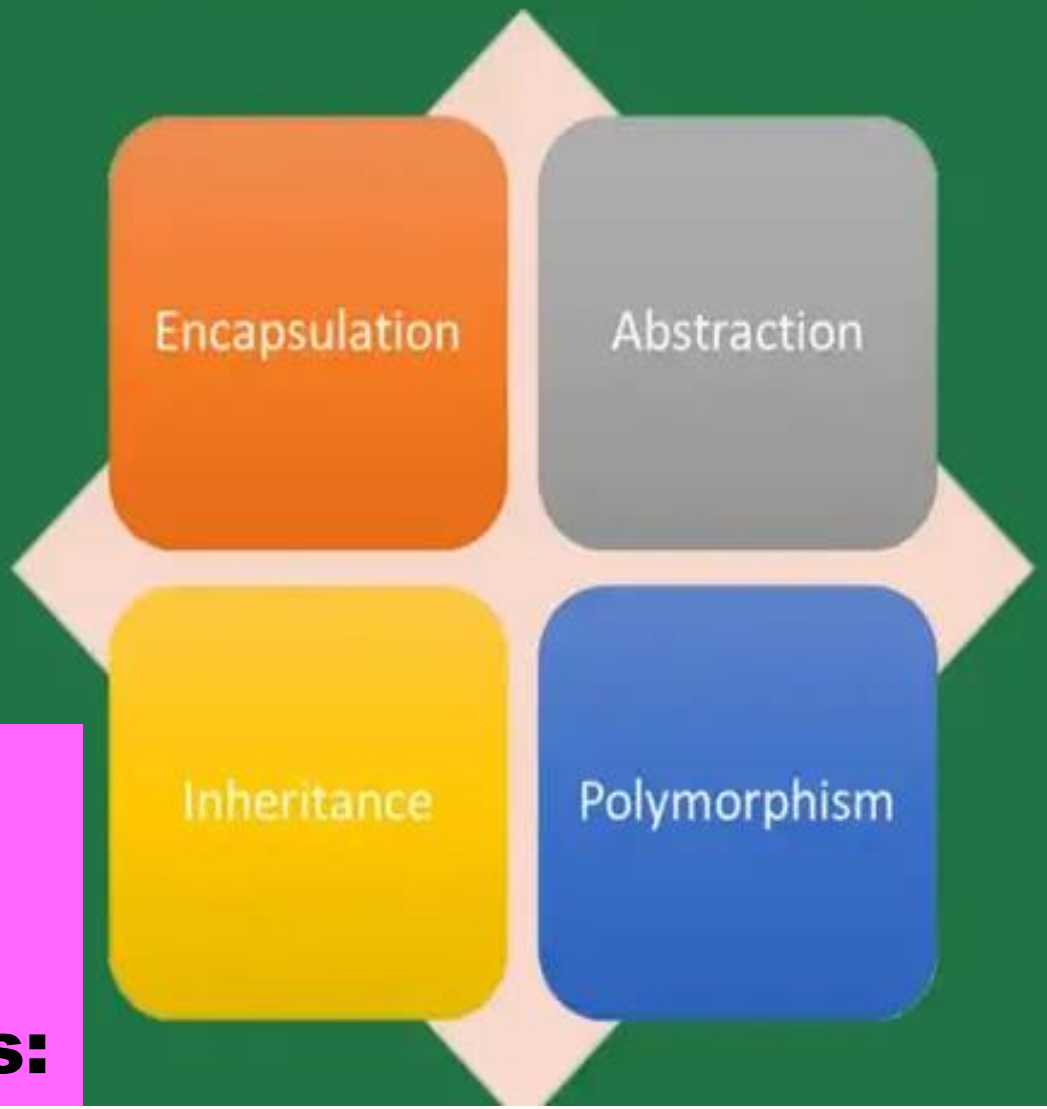
- **Methods are functions that are defined inside a class that describe the behaviors of an object. Each method contained in class definitions starts with a reference to an instance object. Additionally, the subroutines contained in an object are called instance methods. Programmers use methods for reusability or keeping functionality encapsulated inside one object at a time.**

- **Attributes are defined in the class template and represent the state of an object. Objects will have data stored in the attributes field. Class attributes belong to the class itself.**

# Object-oriented programming



What are the main principles of OOP?




**Object-oriented programming is based on the following principles:**

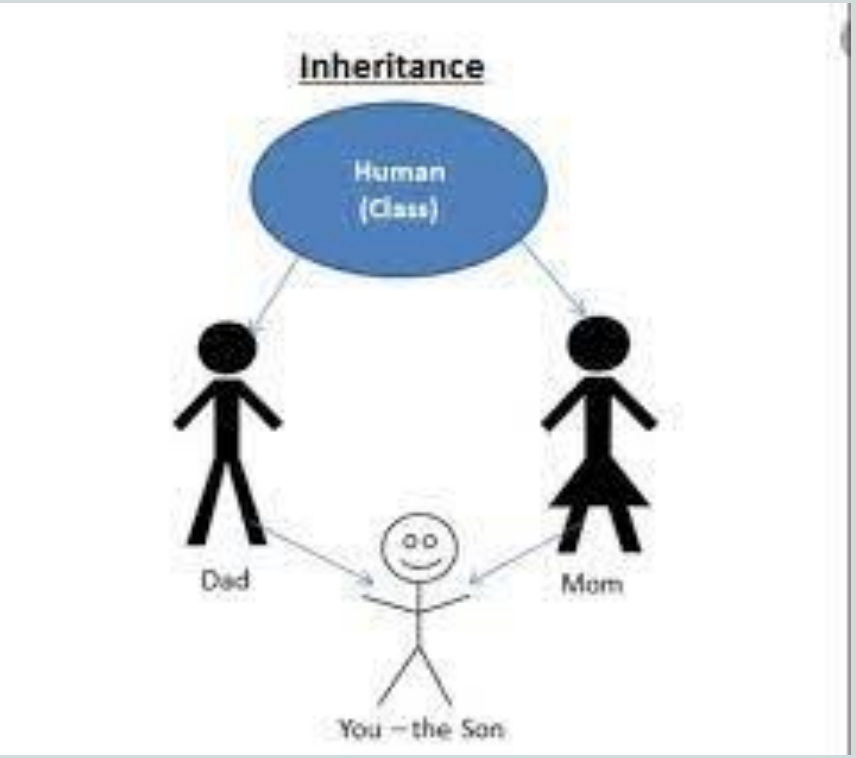
# Encapsulation

Class


**Encapsulation. This principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.**

- **Abstraction. Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers more easily make additional changes or additions over time.**

- **Inheritance. Classes can reuse code from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.**



# Polymorphism


- **Polymorphism. Objects are designed to share behaviors and they can take on more than one form. The program will determine which meaning or usage is necessary for each execution of that object from a parent class, reducing the need to duplicate code. A child class is then created, which extends the functionality of the parent class. Polymorphism allows different types of objects to pass through the same interface.**

Kim Polese, who was the Oak product manager at the time, remembers things differently. "I named Java," she said:

I spent a lot of time and energy on naming Java because I wanted to get precisely the right name. I wanted something that reflected the essence of the technology: dynamic, revolutionary, lively, fun. Because this programming language was so unique, I was determined to avoid nerdy names. I also didn't want anything with 'net' or 'web' in it, because I find those names very forgettable. I wanted something that was cool, unique, and easy to spell and fun to say.

An overview of all Java versions in its history, for Java Standard Edition (SE) Development Kit (JDK).
The latest version of Java is Java 20 or JDK 20 released on March 21st 2023 (follow this article to check Java version on your computer). JDK 20 is a regular update release, and JDK 17 is the most recent Long Term Support (LTS) release of Java SE platform (about 8 years of support from Oracle).
From the first version released in 1996 to the latest version 20 available to the public since March 2023, the Java platform has been actively being developed for more than 27 years. Many changes and improvements have been made to the technology over the years. The following table summarizes all versions of Java SE from its early days to the latest.

| Java SE Version | Version Number | Release Date |
|---|---|---|
| **JDK 1.0** (Oak) | 1.0 | January 1996 |
| **JDK 1.1** | 1.1 | February 1997 |
| **J2SE 1.2** (Playground) | 1.2 | December 1998 |
| **J2SE 1.3** (Kestrel) | 1.3 | May 2000 |
| **J2SE 1.4** (Merlin) | 1.4 | February 2002 |
| **J2SE 5.0** (Tiger) | 1.5 | September 2004 |
| **Java SE 6** (Mustang) | 1.6 | December 2006 |
| **Java SE 7** (Dolphin) | 1.7 | July 2011 |
| **Java SE 8** | 1.8 | March 2014 |
| **Java SE 9** | 9 | September, 21st 2017 |
| **Java SE 10** | 10 | March, 20th 2018 |
| **Java SE 11** | 11 | September, 25th 2018 |
| **Java SE 12** | 12 | March, 19th 2019 |
| **Java SE 13** | 13 | September, 17th 2019 |
| **Java SE 14** | 14 | March, 17th 2020 |
| **Java SE 15** | 15 | September, 15th 2020 |
| **Java SE 16** | 16 | March, 16th 2021 |
| **Java SE 17** | 17 | September, 14th 2021 |
| **Java SE 18** | 18 | March, 22nd 2022 |
| Java SE 19 | 19 | September 20th, 2022 |
| Java SE 20 | 20 | March 21st, 2023 |

https://www.w3schools.com/java/default.asp

# JAVA INTRODUCTION

# Java Programming

**Java is a powerful general-purpose programming language. It is one of the most popular programming languages used to develop desktop and mobile applications, big data processing, embedded systems and so on.**

**It is used for:**

**Mobile applications (specially Android apps)**

**Desktop applications**

**Web applications**

**Web servers and application servers**

**Games**

**Database connection**

**And much, much more!**

What is Java?

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than 3 billion devices run Java.

- **Why Use Java?**

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming language in the world
- It has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has a huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to C++ and C#, it makes it easy for programmers to switch to Java or vice versa

## Features of Java Programming

### Java is platform-independent

Java was built with the philosophy of "write once, run anywhere" (WORA). The Java code you write on one platform (operating system) will run on other platforms with no modification.

### An object-oriented Language

The object-oriented approach is one of the popular programming styles. In object-oriented programming, a complex problem is divided into smaller sets by creating objects. This makes Java code reusable, has design benefits and makes code easier to maintain.

**Java is fast**
**The earlier version of Java was criticized for being slow. However, the new version of Java is one of the fastest programming languages.**

**A well-optimized Java code is nearly as fast as lower-level languages like C/C++ and much faster than Python, PHP, etc.**

**Java is secure**
**Some of the high-level features for security that Java handles are:**

**provides a secure platform for developing and running applications**
**automatic memory management reduces memory corruption and vulnerabilities**

# Large Standard Library

**One of the reasons why Java is widely used is because of the availability of a huge standard library. The Java environment has hundreds of classes and methods under different packages to help software developers like us. For example,**

**java.lang- for advanced features of strings, arrays, etc**
**java.util - for data structures, regular expressions, date and time functions, etc**
**java.io - for file i/o, exception handling, etc**

## Applications of Java Programming

According to Oracle, the company that owns Java, Java runs on 3 billion devices worldwide, which makes Java one of the most popular programming languages.

### 1. Android apps

Java programming language using Android SDK (Software Development Kit) is usually used for developing Android apps.

### 2. Web apps

Java is used to create Web applications through Servlets, Struts or JSPs. Some of the popular web applications written in Java are Google.com, Facebook.com, eBay.com, LinkedIn.com, etc.

### 3. Big Data Processing

You can use a popular software framework like Hadoop (which is written in Java) to process Big Data.

### 4. Embedded Devices

Oracle's Java Embedded technologies provide a platform and runtime for billions of embedded devices like televisions, SIM cards, Blu-ray Disc players, etc.

Besides these applications, Java is also used for game development, scientific applications (like natural language processing), and many others.

## What Is the JVM?

A Virtual Machine is a software implementation of a physical machine. Java was developed with the concept of WORA (Write Once Run Anywhere), which runs on a VM. The compiler compiles the Java file into a Java .class file, then that .class file is input into the JVM, which loads and executes the class file.

**Java JDK, JRE and JVM**

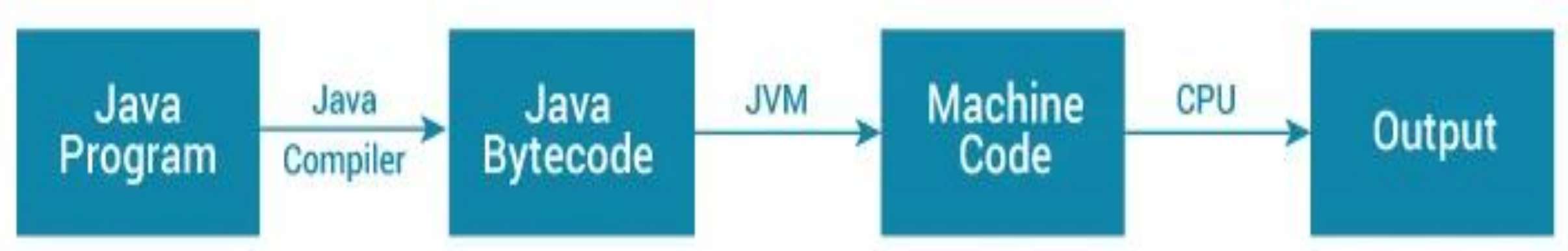
**What is JVM?**

**JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.**

**When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).**

**Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.**

## Working of Java Program




**What is 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.**

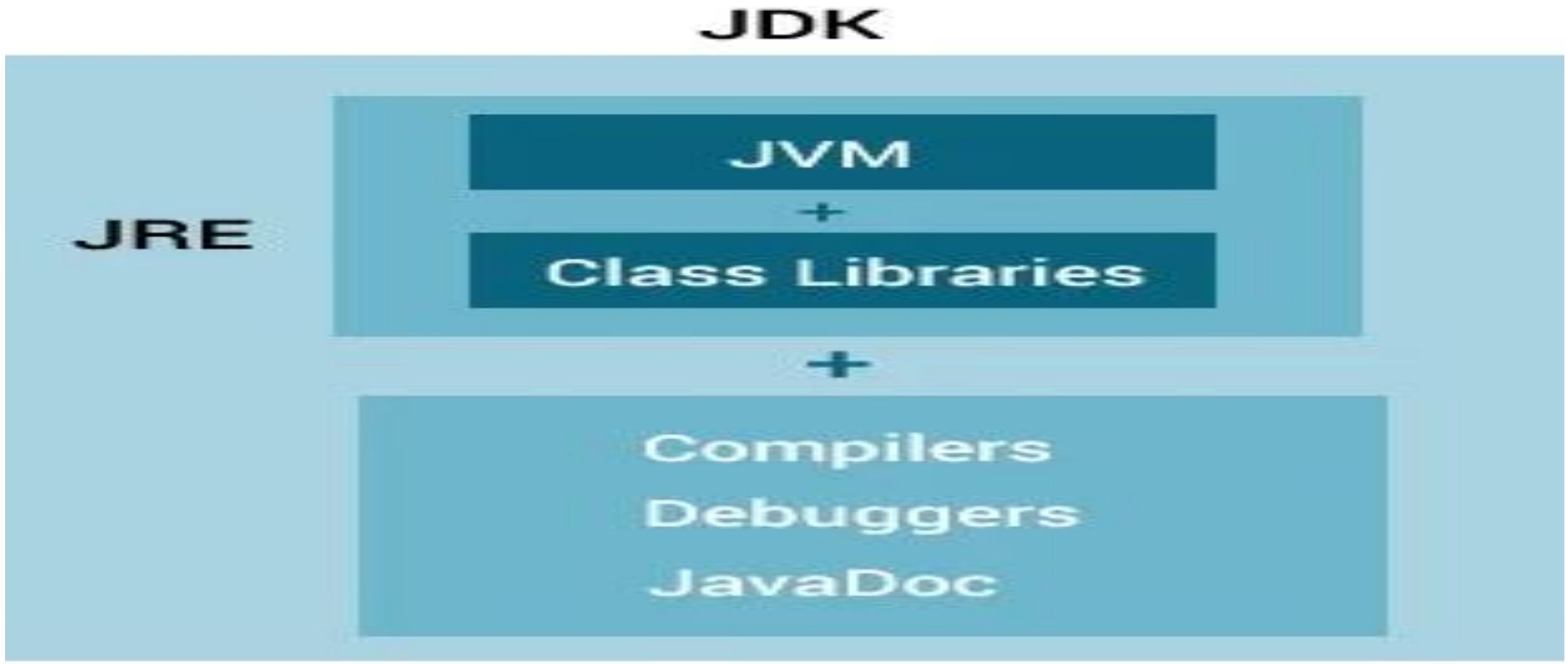
**What is JDK?**

**JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.**

**In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).**


**Java Development Kit**

**JDK contains JRE and other tools to develop Java applications.**




Relationship between JVM, JRE, and JDK

**JRE contains JVM and class libraries and JDK contains JRE, compilers, debuggers, and JavaDoc**

**Java Install**
**Some PCs might have Java already installed.**

**To check if you have Java installed on a Windows PC, search in the start bar for Java or type the following in Command Prompt (cmd.exe):**

**C:\Users\Your Name>java -version**
**If Java is installed, you will see something like this (depending on version):**

**java version "11.0.1" 2018-10-16 LTS**
**Java(TM) SE Runtime Environment 18.9 (build 11.0.1+13-LTS)**
**Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.1+13-LTS, mixed mode)**
**If you do not have Java installed on your computer, you can download it for free at oracle.com.**

**We will write Java code in a text editor. However, it is possible to write Java in an Integrated Development Environment, such as IntelliJ IDEA, Netbeans or Eclipse, which are particularly useful when managing larger collections of Java files.**

**your machine**

**Setup for Windows**
**To install Java on Windows:**

**Go to "System Properties" (Can be found on Control Panel > System and Security > System > Advanced System Settings)**
**Click on the "Environment variables" button under the "Advanced" tab**
**Then, select the "Path" variable in System variables and click on the "Edit" button**
**Click on the "New" button and add the path where Java is installed, followed by \bin. By default, Java is installed in C:\Program Files\Java\jdk-11.0.1 (If nothing else was specified when you installed it). In that case, You will have to add a new path with: C:\Program Files\Java\jdk-11.0.1\bin**
**Then, click "OK", and save the settings**

**Open Command Prompt (cmd.exe) and type java -version to see if Java is running on**

**Java Quickstart**

**In Java, every application begins with a class name, and that class must match the filename.**

**Let's create our first Java file, called Main.java, which can be done in any text editor (like Notepad).**

**The file should contain a "Hello World" message, which is written with the following code:**


Java Syntax
main() method
Class & Object
Keywords
Package
Literals
Arrays
Methods
Variables
Interfaces
Data types

# Java Syntax

## Main.java

```
public class Main {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

# JAVA OUTPUT / PRINT

- **Print Text**
- **you can use the println() method to output values or print text in Java**:

You can add as many println() methods as you want. Note that it will add a new line for each method:

Example

```
System.out.println("Hello World!");
System.out.println("I am learning Java.");
System.out.println("It is awesome!");
```

Double Quotes

When you are working with text, it must be wrapped inside double quotations marks "".

If you forget the double quotes, an error occurs:

# JAVA OUTPUT NUMBERS

You can also perform mathematical calculations inside the println() method:

Example

```
System.out.println(3 + 3);
```

```
System.out.println(2 * 5);
```

- **Print Numbers**
- **You can also use the println() method to print numbers.**

- **However, unlike text, we don't put numbers inside double quotes:**

- **Example**
- **System.out.println(3);**
- **System.out.println(358);**
- **System.out.println(50000);**

# JAVA COMMENTS

- Java Comments
- Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

- Single-line Comments
- Single-line comments start with two forward slashes (//).

- Any text between // and the end of the line is ignored by Java (will not be executed).

This example uses a single-line comment before a line of code:

Example

```
// This is a comment
System.out.println("Hello World");
```

This example uses a single-line comment at the end of a line of code:

Example

```
System.out.println("Hello World"); // This is a comment
```

- Java Multi-line Comments
- Multi-line comments start with /* and ends with */.
- Any text between /* and */ will be ignored by Java.
- This example uses a multi-line comment (a comment block) to explain the code:
- Example
- /* The code below will print the words Hello World
- to the screen, and it is amazing */
- System.out.println("Hello World");

Single or multi-line comments?

It is up to you which you want to use. Normally, we use // for short comments, and /* */ for longer.

# JAVA IDENTIFIERS, RESERVED KEYWORDS AND CONTROL STATEMENTS

**Identifiers in Java are one of the basic fundamentals of Java that is mandatory for any Java learner. Without learning the identifiers, its rules and naming convention, you can’t efficiently program in Java. The naming conventions are optional, but you should follow them as a rule so that it increases the readability of the code.**

**- it is a variable name in java.**

## Keywords

- also are known as reserved words are the pre-defined identifiers reserved by Java for a specific purpose that inform the compiler about what the program should do.
-have a special meaning those already explained to the java language like int, float, class, public, etc. these are the reserved keywords.
-These special words cannot be used as class names, variables, or method names, because they have special meaning within the language.

Control Statements in Java is one of the fundamentals required for Java Programming. It allows the smooth flow of a program.

# List of Java Keywords

## Primitive Types and void

1. boolean
2. byte
3. char
4. short
5. int
6. long
7. float
8. double
9. void

## Modifiers

1. public
2. protected
3. private
4. abstract
5. static
6. final
7. transient
8. volatile
9. synchronized
10. native

## Declarations

1. class
2. interface
3. enum
4. extends
5. implements
6. package
7. throws

## Control Flow

1. if
2. else
3. try
4. catch
5. finally
6. do
7. while
8. for
9. continue
10. break
11. switch
12. case
13. default

## Miscellaneous

1. this
2. new
3. super
4. import
5. instanceof
6. null
7. true
8. false
9. strictfp
10. assert
11. _ (underscore)
12. goto
13. const

# Java Variables

Variables are containers for storing data values.

**Different types of variables:**

**String** - stores text, such as "Hello". String values are surrounded by double quotes

**Int** - stores integers (whole numbers), without decimals, such as 123 or -123

**float** - stores floating point numbers, with decimals, such as 19.99 or -19.99

**char** - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes

**boolean** - stores values with two states: true or false

# Declaring (Creating) Variables

## Syntax

```
type variableName = value;
```

## Example

Create a variable called name of type String and assign it the value "John":

```
public class Main {
  public static void main(String[] args) {
    String name = "John";
    System.out.println(name);
  }
}
```

```
String name = "John";
System.out.println(name);
```

Example
Create a variable called myNum of type int and assign it the value 15:

```
int myNum = 15;
System.out.println(myNum);
```

```
public class Main {
  public static void main(String[] args) {
    int myNum = 15;
    System.out.println(myNum);
  }
}
```

Example
Change the value of myNum from 15 to 20:

```
int myNum = 15;
myNum = 20;  // myNum is now 20
System.out.println(myNum);
```

## Final Variables

If you don't want others (or yourself) to overwrite existing values, use the final keyword (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

**Example**

```
public class Main {
  public static void main(String[] args) {
    final int myNum = 15;
    myNum = 20; // will generate an error
    System.out.println(myNum);
  }
}
```

```
final int myNum = 15;
myNum = 20;  // will generate an error: cannot assign a value to a final
```

# Other Types

## Example

```
int myNum = 5;
float myFloatNum = 5.99f;
```

```
char myLetter = 'D';
boolean myBool = true;
String myText = "Hello";
```

# JAVA PRINT VARIABLES

## Display Variables

The println() method is often used to display variables.

To combine both text and a variable, use the + character:

Example

String name = "John";

System.out.println("Hello " + name);

```
public class Main {
  public static void main(String[] args) {
    String name = "John";
    System.out.println("Hello " + name);
  }
}
```

**For numeric values, the + character works as a mathematical operator (notice that we use int (integer) variables here):**

**Example**

```
int x = 5;
int y = 6;
System.out.println(x + y); // Print the value of x + y
```

**x stores the value 5**
**y stores the value 6**
**Then we use the println() method to display the value of x + y, which is 11**

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    int y = 6;
    System.out.println(x + y); // Print the value of x + y
  }
}
```

# Java Declare Multiple Variables

## Declare Many Variables

To declare more than one variable of the same type, you can use a comma-separated list:

Instead of writing:

```
int x = 5;
int y = 6;
int z = 50;
System.out.println(x + y + z);
```

You can simply write:

```
int x = 5, y = 6, z = 50;
System.out.println(x + y + z);
```

```
public class Main {
  public static void main(String[] args) {
    int x = 5, y = 6, z = 50;
    System.out.println(x + y + z);
  }
}
```

# One Value to Multiple Variables

You can also assign the same value to multiple variables in one line:

```
int x, y, z;
x = y = z = 50;
System.out.println(x + y + z);
```

```
public class Main {
  public static void main(String[]
args) {
    int x, y, z;
    x = y = z = 50;
    System.out.println(x + y + z);
  }
}
```

# Java Identifiers

**Identifiers**
**All Java variables must be identified with unique names.**

**These unique names are called identifiers.**

**Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).**

# Example

```
// Good
int minutesPerHour = 60;

// OK, but not so easy to understand what m actually is
int m = 60;
```

```
public class Main {
  public static void main(String[] args) {
    // Good
    int minutesPerHour = 60;

    // OK, but not so easy to understand
what m actually is
    int m = 60;

    System.out.println(minutesPerHour);
    System.out.println(m);
  }
}
```

**The general rules for naming variables are:**

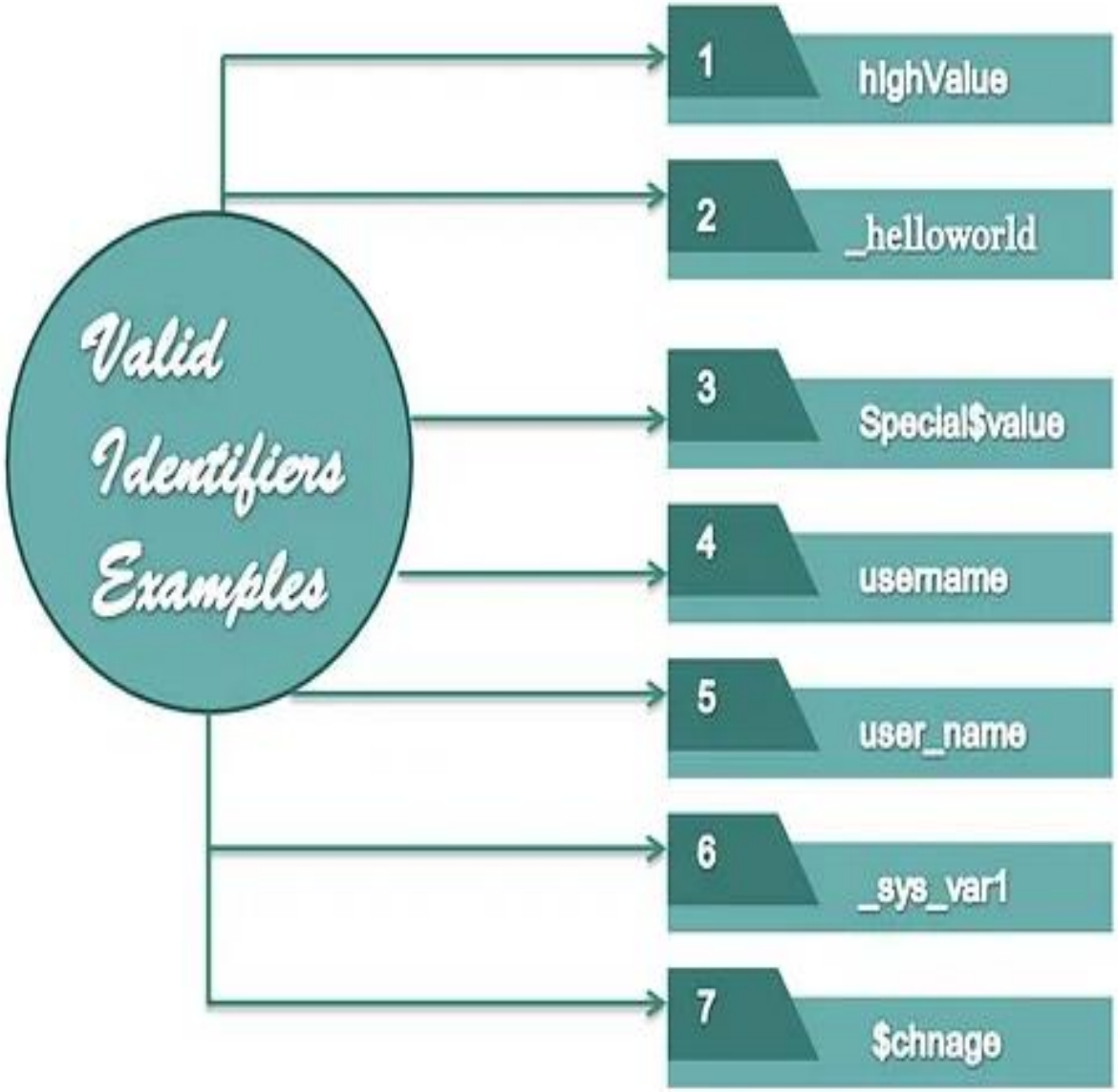
**Names can contain letters, digits, underscores, and dollar signs**

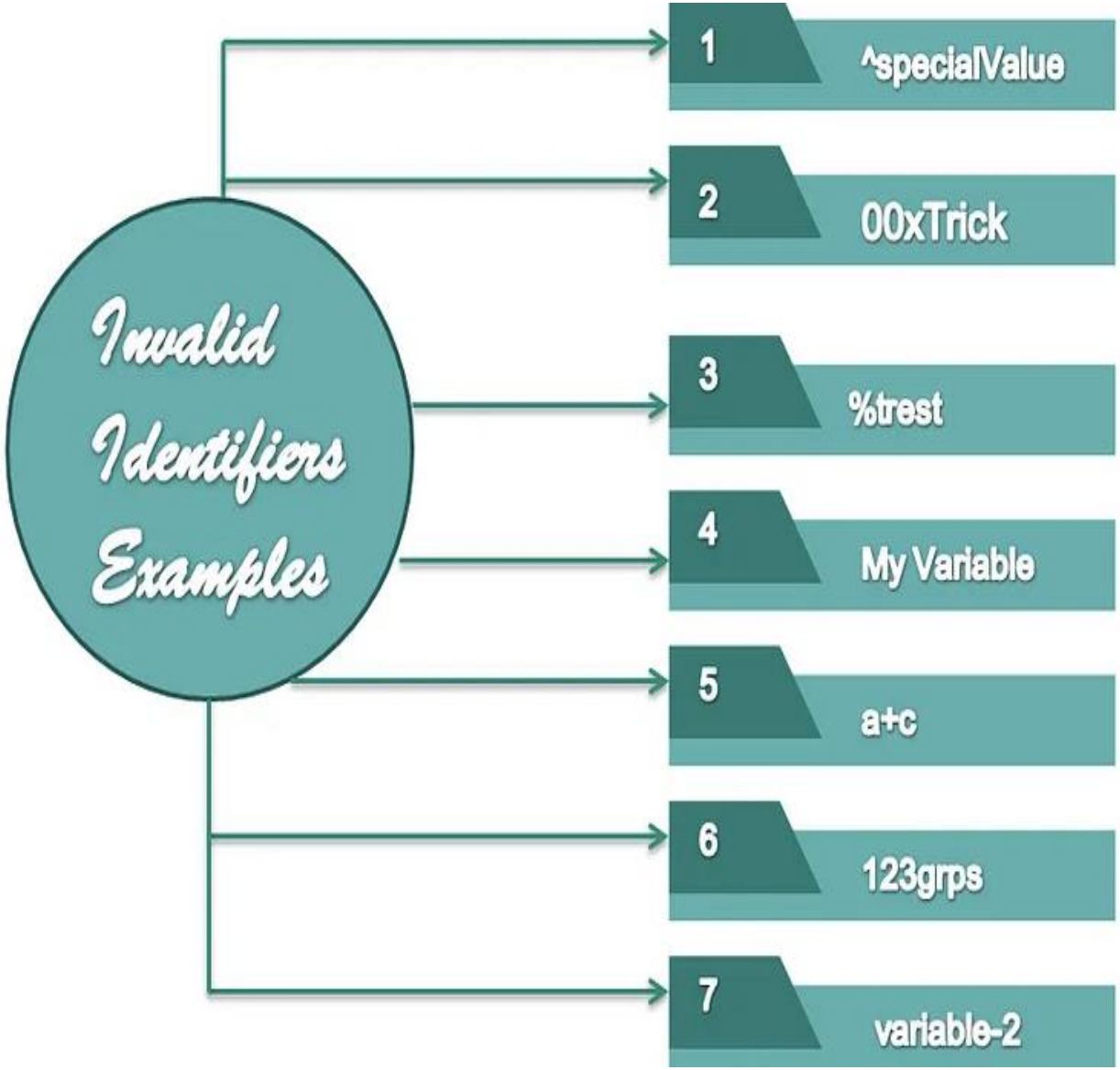
**Note: It is recommended to use descriptive names in order to create understandable and maintainable code:**

**different variables)**

**Reserved words (like Java keywords, such as int or boolean) cannot be used as names**

# Valid and Invalid Identifiers



# Java Data Types

**A variable in Java must be a specified data type:**

```
int myNum = 5;          // Integer (whole number)
```

```
float myFloatNum = 5.99f;    // Floating point number
```

```
char myLetter = 'D';     // Character
```

```
boolean myBool = true;          // Boolean
```

```
String myText = "Hello";          // String
```

```
public class Main {
  public static void main(String[] args) {
    int myNum = 5;                  // integer (whole number)
    float myFloatNum = 5.99f;    // floating point number
    char myLetter = 'D';          // character
    boolean myBool = true;        // boolean
    String myText = "Hello";      // String
    System.out.println(myNum);
    System.out.println(myFloatNum);
    System.out.println(myLetter);
    System.out.println(myBool);
    System.out.println(myText);
  }
}
```

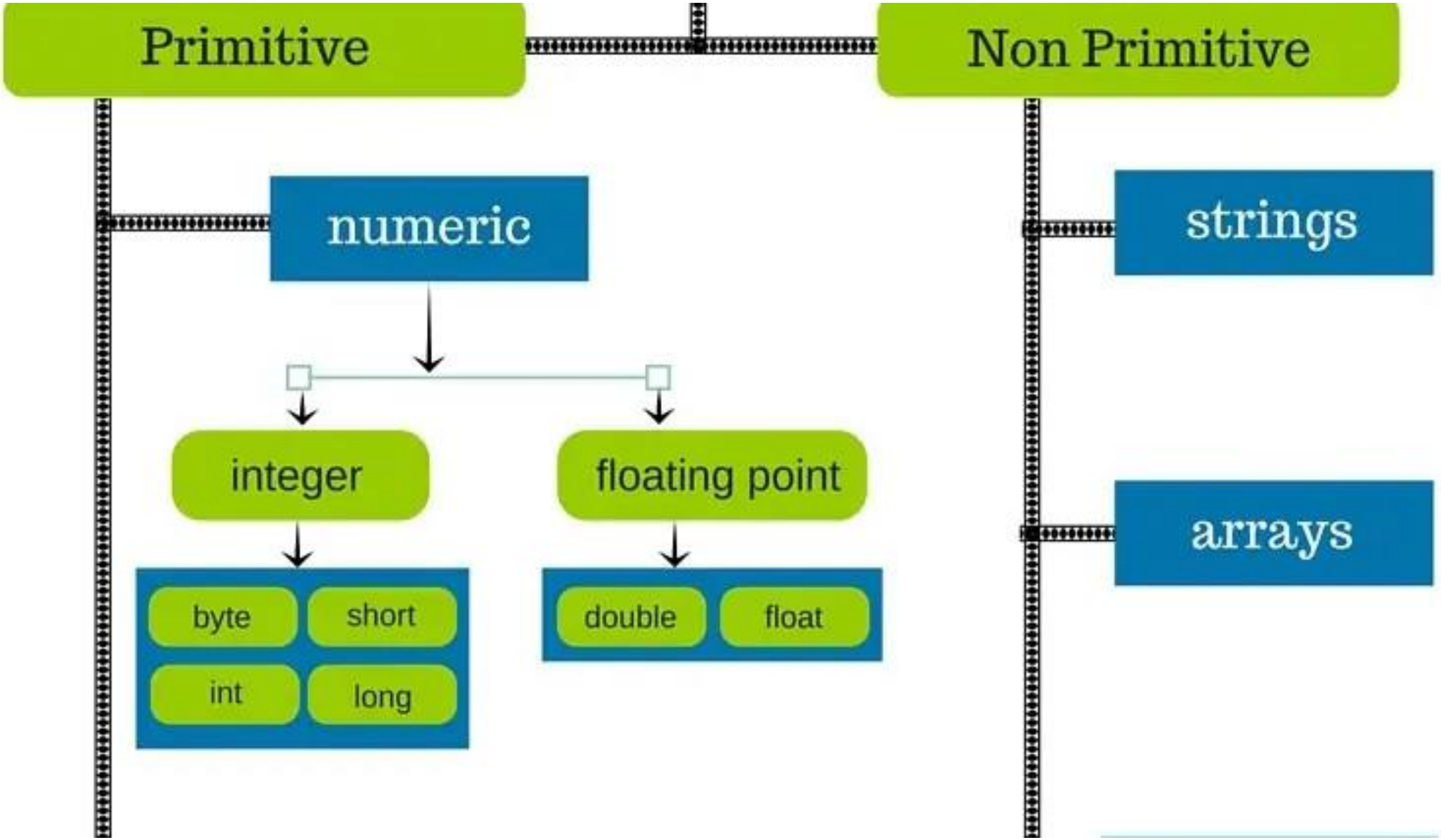
## Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

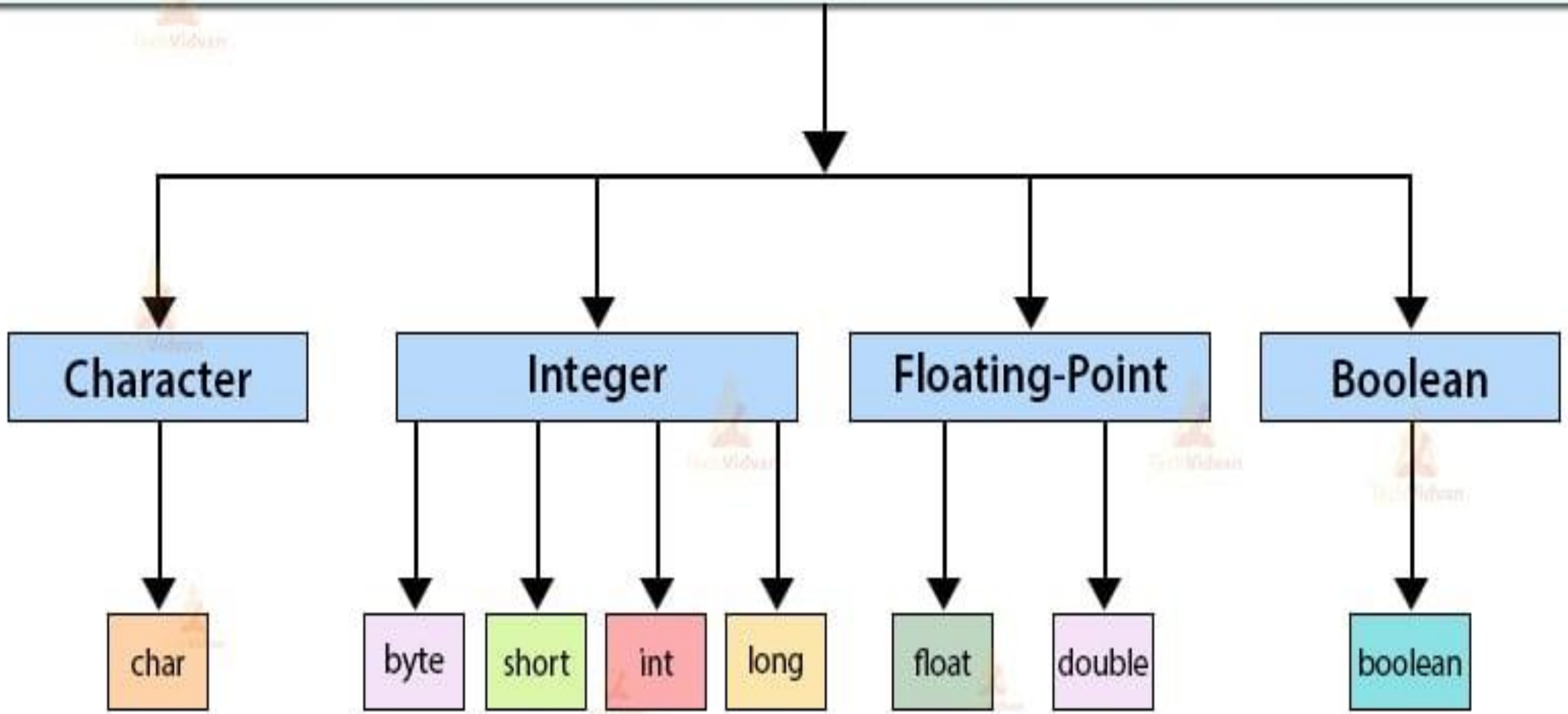
There are eight primitive data types in Java:

| Data Type | Size | Description |
|---|---|---|
| byte | 1 byte | Stores whole numbers from -128 to 127 |
| short | 2 bytes | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| boolean | 1 bit | Stores true or false values |
| char | 2 bytes | Stores a single character/letter or ASCII values |

# Data types are divided into two groups:



# Primitive Data Types in Java



# Primitive number types are divided into two groups:

## JAVA NUMBERS

**Integer types**

-stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long. Which type you should use, depends on the numeric value.

**Floating point types**

- represents numbers with a fractional part, containing one or more decimals. There are two types: float and double.

# Integer Types

## Byte

The byte data type can store whole numbers from -128 to 127. This can be used instead of int or other integer types to save memory when you are certain that the value will be within -128 and 127:

Example

```
byte myNum = 100;
System.out.println(myNum);
```

```
public class Main {
  public static void
main(String[] args) {
    byte myNum = 100;

System.out.println(m
yNum);
  }
}
```

## Short

The short data type can store whole numbers from
-32768 to 32767:

Example
short myNum = 5000;
System.out.println(myNum);

```
public class Main {
  public static void main(String[]
args) {
    short myNum = 5000;
    System.out.println(myNum);
  }
}
```

## Int

The int data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

Example
int myNum = 100000;
System.out.println(myNum);

```
public class Main {
  public static void
main(String[] args) {
    int myNum = 100000;

System.out.println(myNum);
  }
}
```

## Long

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

**Example**

```
long myNum = 15000000000L;
System.out.println(myNum);
```

```
public class Main {
  public static void main(String[] args) {
    long myNum = 15000000000L;
    System.out.println(myNum);
  }
}
```

## Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

The float and double data types can store fractional numbers. Note that you should end the value with an "f" for floats and "d" for doubles:

## Float

**Example**

```
float myNum = 5.75f;
System.out.println(myNum);
```

```
public class Main {
  public static void main(String[] args) {
    float myNum = 5.75f;

System.out.println(myNum);
  }
}
```

## Double

**Example**

```
double myNum = 19.99d;
System.out.println(myNum);
```

```
public class Main {
  public static void main(String[]
args) {
    double myNum = 19.99d;
    System.out.println(myNum);
  }
}
```

## Use float or double?

The precision of a floating point value indicates how many digits the value can have after the decimal point. The precision of float is only six or seven decimal digits, while double variables have a precision of about 15 digits. Therefore it is safer to use double for most calculations.

## Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

Example

```
float f1 = 35e3f;
double d1 = 12E4d;
System.out.println(f1);
System.out.println(d1);
```

```
public class Main {
  public static void
main(String[] args) {
    float f1 = 35e3f;
    double d1 = 12E4d;
    System.out.println(f1);
    System.out.println(d1);
  }
}
```

# JAVA BOOLEAN DATA TYPES

## Boolean Types

Very often in programming, you will need a data type that can only have one of two values, like:

YES / NO
ON / OFF
TRUE / FALSE

For this, Java has a boolean data type, which can only take the values true or false:

**Example**

```
boolean isJavaFun = true;
boolean isFishTasty = false;
System.out.println(isJavaFun);      // Outputs true
System.out.println(isFishTasty);    // Outputs false
```

```
public class Main {
  public static void main(String[] args) {
    boolean isJavaFun = true;
    boolean isFishTasty = false;
    System.out.println(isJavaFun);
    System.out.println(isFishTasty);
  }
}
```

Boolean values are mostly used for conditional testing.

# JAVA CHARACTERS

The char data type is used to store a single character. The character must be surrounded by single quotes, like 'A' or 'c':

Example

```
char myGrade = 'B';
System.out.println(myGrade);
```

Alternatively, if you are familiar with ASCII values, you can use those to display certain characters:

Example

```
char myVar1 = 65, myVar2 = 66, myVar3 = 67;
System.out.println(myVar1);
System.out.println(myVar2);
System.out.println(myVar3);
```

A list of all ASCII values can be found in our ASCII Table Reference.

```
public class Main {
  public static void main(String[] args) {
    char myGrade = 'B';
    System.out.println(myGrade);
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    char myVar1 = 65, myVar2 = 66, myVar3 = 67;
    System.out.println(myVar1);
    System.out.println(myVar2);
    System.out.println(myVar3);
  }
}
```

# Strings

The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
String greeting = "Hello World";
System.out.println(greeting);
```

The String type is so much used and integrated in Java, that some call it "the special ninth type".

A String in Java is actually a non-primitive data type, because it refers to an object. The String object has methods that are used to perform certain operations on strings.

```
public class Main {
  public static void main(String[] args) {
    String greeting = "Hello World";
    System.out.println(greeting);
  }
}
```

# JAVA NON-PRIMITIVE DATA TYPES

## Non-Primitive Data Types

**Non-primitive data types are called reference types because they refer to objects.**

**The main difference between primitive and non-primitive data types are:**

- **Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).**
- **Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.**
- **A primitive type has always a value, while non-primitive types can be null.**
- **A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.**

**Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc.**

# JAVA TYPE CASTING

Type casting is when you assign a value of one primitive data type to another type.

## In Java, there are two types of casting:

**Widening Casting (automatically) - converting a smaller type to a larger type size**

**byte -> short -> char -> int -> long -> float -> double**

**Narrowing Casting (manually) - converting a larger type to a smaller size type**

**double -> float -> long -> int -> char -> short -> byte**

### Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

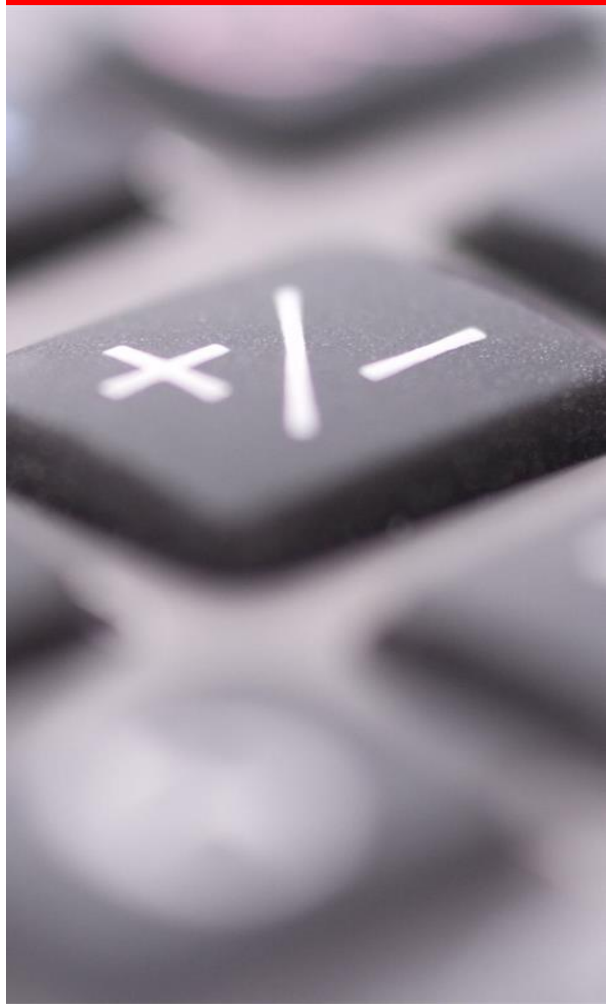
Example

```
public class Main {
  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
  }
```

```
public class Main {
  public static void main(String[] args) {
    int myInt = 9;
    double myDouble = myInt; // Automatic casting: int to double

    System.out.println(myInt);
    System.out.println(myDouble);
  }
}
```

JAVA TYPE CASTING



# Narrowing Casting

**Narrowing casting must be done manually by placing the type in parentheses in front of the value:**

# Example

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78d;
    int myInt = (int) myDouble; // Manual casting: double to int

    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

# OPERATORS IN JAVA

| Operators Type | Operators |
|---|---|
| Arithmetic | +, -, *, /, %, ++, --, +=, -=, *=, /=, %= |
| Bitwise | ~, &, \|, ^, >>, >>>, <<, &=, \|=, ^=, >>=, >>>=, <<= |
| Relational | ==, !=, >, <, >=, <= |
| Logical | &, \|, ^, \|\|, &&, !, ==, &=, \|=, ^=, !=, ?: |
| Assignment | = |

# Java Operators

**Operators are used to perform operations on variables and values.**
**In the example below, we use the + operator to add together two values:**

```
public class Main {
  public static void main(String[] args) {
    int sum1 = 100 + 50;
    int sum2 = sum1 + 250;
    int sum3 = sum2 + sum2;
    System.out.println(sum1);
    System.out.println(sum2);
    System.out.println(sum3);
  }
}
```

## Example

```
int sum1 = 100 + 50;        // 150 (100 + 50)
int sum2 = sum1 + 250;      // 400 (150 + 250)
int sum3 = sum2 + sum2;     // 800 (400 + 400)
```

```
public class Main {
  public static void main(String[] args) {
    int x = 100 + 50;
    System.out.println(x);
  }
}
```

Example
int x = 100 + 50;
Although the + operator is often used to add together two values, like in the example above, it can also be used to add together a variable and a value, or a variable and another variable:

# Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Logical operators
- Bitwise operators
- Comparison operators

## Arithmetic operators

-are used to perform common mathematical operations.

| Operator | Name | Description | Example |
|---|---|---|---|
| + | Addition | Adds together two values | x + y |
| - | Subtraction | Subtracts one value from another | x - y |
| * | Multiplication | Multiplies two values | x * y |
| / | Division | Divides one value by another | x / y |
| % | Modulus | Returns the division remainder | x % y |
| ++ | Increment | Increases the value of a variable by 1 | ++x |
| -- | Decrement | Decreases the value of a variable by 1 | --x |

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    int y = 3;
    System.out.println(x + y);
    System.out.println(x-y);
    System.out.println(x * y);
    System.out.println(x / y);
    System.out.println(x%y);

  }
}
```

# Java Assignment Operators

Assignment operators are used to assign values to variables.

below, we use the assignment operator (=) to assign the value 10 to a variable

Example

int x = 10;

| Operator | Operation | Equivalent to |
|---|---|---|
| = | num = 5 | num = 5 |
| += | num+=5 | num = num+5 |
| -= | num-=5 | num = num-5 |
| *= | num*=5 | num = num*5 |
| /= | num/=5 | num = num/5 |

| Operator | Example | Same As |
|---|---|---|
| %= | x %= 3 | x = x % 3 |
| &= | x &= 3 | x = x & 3 |
| \|= | x \|= 3 | x = x \| 3 |
| ^= | x ^= 3 | x = x ^ 3 |
| >>= | x >>= 3 | x = x >> 3 |
| <<= | x <<= 3 | x = x << 3 |

# Java Comparison Operators

**Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.**

**The return value of a comparison is either true or false. These values are known as Boolean values.**

**Example**

```
int x = 5;
int y = 3;
```

```
System.out.println(x > y); // returns true, because 5 is higher than 3
```

| Operator | Name | Example |
|---|---|---|
| == | Equal to | x == y |
| != | Not equal | x != y |
| > | Greater than | x > y |
| < | Less than | x < y |
| >= | Greater than or equal to | x >= y |
| <= | Less than or equal to | x <= y |

# Java Logical Operators

- are used to determine the logic between variables or values (test for true or false of a value)

## Logical Operators in Java

1. Logical AND Operator (& and &&)

| Operand 1 | Operand 2 | Returned Value |
|---|---|---|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

2. Logical OR Operator (| and ||)

| Operand 1 | Operand 2 | Returned Value |
|---|---|---|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

3. Logical NOT Operator (!)

| Operand | Returned Value |
|---|---|
| False | True |
| True | False |

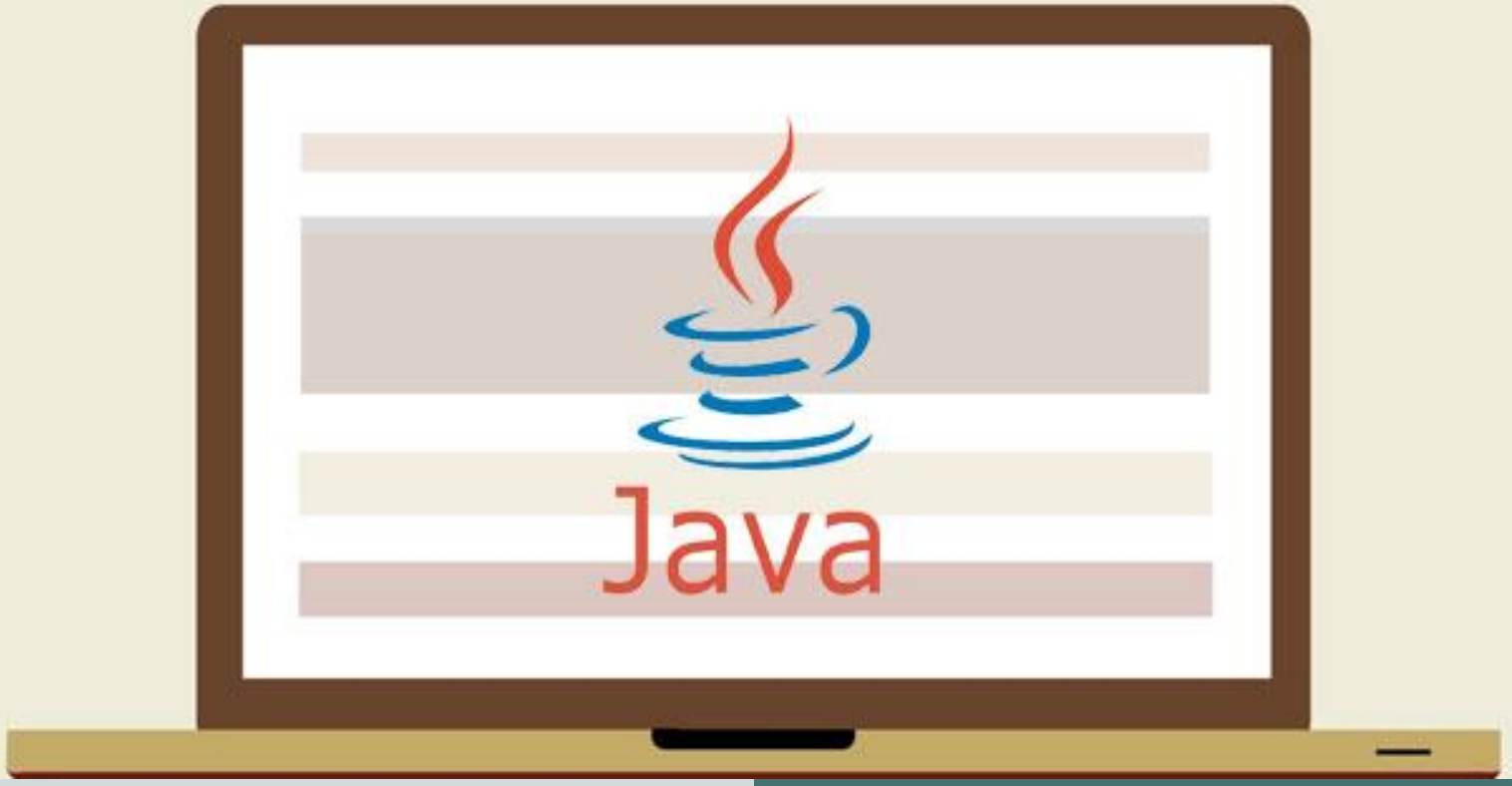
**Example**

x < 5 && x < 10

x < 5 || x < 4

!(x < 5 && x < 10)

# Java String Operators



# Java Strings

Strings are used for storing text.

A String variable contains a collection of characters surrounded by double quotes:

Example
Create a variable of type String and assign it a value:

```
String greeting = "Hello";
```

## String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length() method:

Example

```
String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
System.out.println("The length of the txt string is: " + txt.length());
```

```
public class Main {
  public static void main(String[] args) {
    String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    System.out.println("The length of the txt string is: "
+ txt.length());
  }
}
```

## More String Methods

**There are many string methods available, for example toUpperCase() and toLowerCase():**

**Example**

```
String txt = "Hello World";
System.out.println(txt.toUpperCase());   // Outputs "HELLO WORLD"
System.out.println(txt.toLowerCase());   // Outputs "hello world"
```

```
public class Main {
  public static void main(String[] args) {
    String txt = "Hello World";
    System.out.println(txt.toUpperCase());
    System.out.println(txt.toLowerCase());
  }
}
```

## Finding a Character in a String

The indexOf() method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace):

Example

```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

Java counts positions from zero.
0 is the first position in a string, 1 is the second, 2 is the third ...

Complete String Reference
For a complete reference of String methods, go to our Java String Methods Reference.

The reference contains descriptions and examples of all string methods.

```
public class Main {
  public static void main(String[] args) {
    String txt = "Please locate where 'locate' occurs!";
    System.out.println(txt.indexOf("locate"));
  }
}
```

## Java String Concatenation

The + operator can be used between strings to combine them. This is called concatenation:

Example

```
String firstName = "John";
String lastName = "Doe";
System.out.println(firstName + " " + lastName);
```

```
public class Main {
  public static void main(String args[]) {
    String firstName = "John";
    String lastName = "Doe";
    System.out.println(firstName + " " +
lastName);
  }
}
```

Note that we have added an empty text (" ") to create a space between firstName and lastName on print.

**concat() method to concatenate two strings:**

**Example**

```
String firstName = "John ";
String lastName = "Doe";
System.out.println(firstName.concat(lastName));
```

```
public class Main {
  public static void main(String[] args) {
    String firstName = "John ";
    String lastName = "Doe";

System.out.println(firstName.concat(lastName));
  }
}
```

# Java Numbers and Strings

## Adding Numbers and Strings

**WARNING!**

**Java uses the + operator for both addition and concatenation. Numbers are added. Strings are concatenated.**

**If you add two numbers, the result will be a number:**

**Example**

```
int x = 10;
int y = 20;
int z = x + y;  // z will be 30 (an integer/number)
```

**If you add two strings, the result will be a string concatenation:**

**Example**

```
String x = "10";
String y = "20";
String z = x + y;  // z will be 1020 (a String)
```

```
public class Main {
  public static void main(String[] args) {
    String x = "10";
    String y = "20";
    String z = x + y;
    System.out.println(z);
  }
}
```

## Adding Numbers and Strings

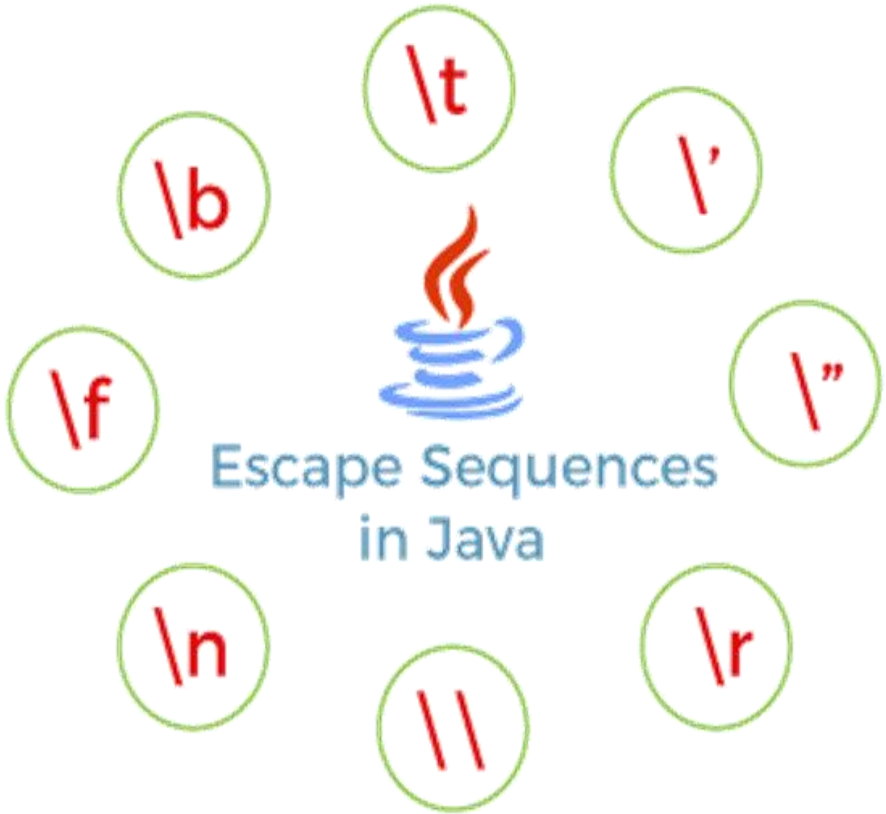
**If you add a number and a string, the result will be a string concatenation:**

**Example**

```
String x = "10";
int y = 20;
String z = x + y;  // z will be 1020 (a String)
```

```
public class Main {
  public static void main(String[] args) {
    String x = "10";
    int y = 20;
    String z = x + y;

System.out.println(z);
  }
}
```




| | |
|---|---|
| \t | Inserts a tab |
| \b | Inserts a backspace |
| \n | Inserts a newline |
| \r | carriage return. () |
| \f | form feed |
| \' | Inserts a single quote |
| \" | Inserts a double quote |
| \\ | Inserts a backslash |

List of Escape Sequences in Java

# Java Special Characters

## Strings - Special Characters

**Because strings must be written within quotes, Java will misunderstand this string, and generate an error:**

**String txt = "We are the so-called "Vikings" from the north.";**

**The solution to avoid this problem, is to use the backslash escape character.**

**The backslash (\) escape character turns special characters into string characters:**

| | |
|---|---|
| \" | Inserts a double quote |

```
public class Main {
  public static void main(String[] args) {
    String txt = "We are the so-called
\"Vikings\" from the north.";
    System.out.println(txt);
  }
}
```

| | |
|---|---|
| \' | Inserts a single quote |

```
public class Main {
  public static void main(String[] args) {
    String txt = "It\'s alright.";
    System.out.println(txt);
  }
}
```

\b Inserts a backspace

```
public class Main {
  public static void main(String[]
args) {
    String txt = "Hel\blo World!";
    System.out.println(txt);
  }
}
```

\r carriage return. ()

```
public class Main {
  public static void main(String[]
args) {
    String txt = "Hello\rWorld!";
    System.out.println(txt);
  }
}
```

\\ Inserts a backslash

```
public class Main {
  public static void main(String[] args) {
                    character \\ is called
    System.out.println(txt
  }
}
```

\t Inserts a tab

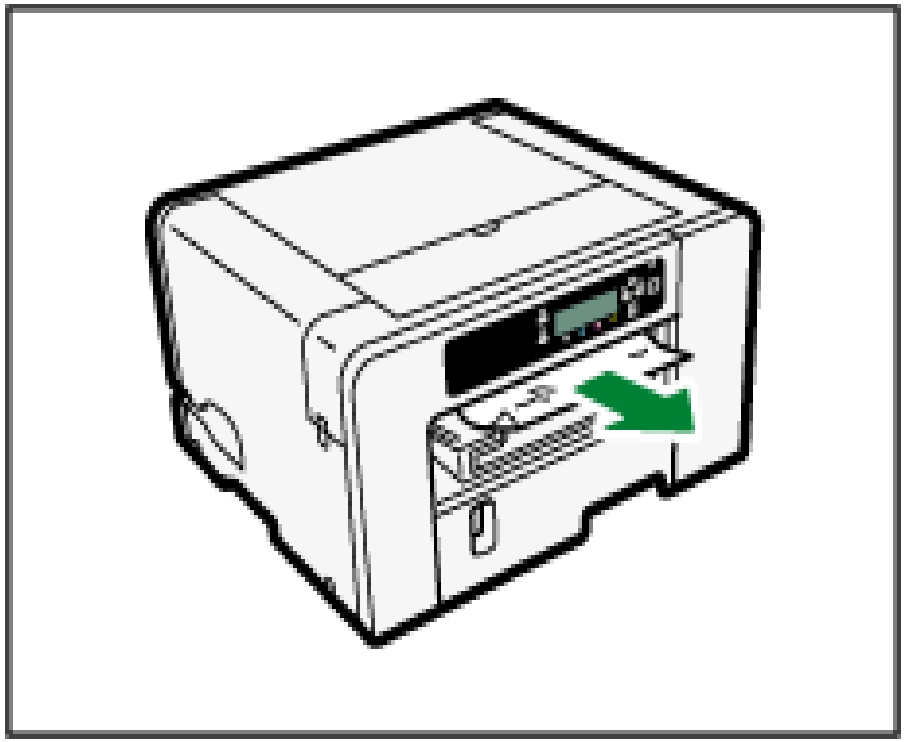
```
public class Main {
  public static void main(String[]
args) {
    String txt = "Hello\tWorld!";
    System.out.println(txt);
  }
}
```

\n Inserts a newline

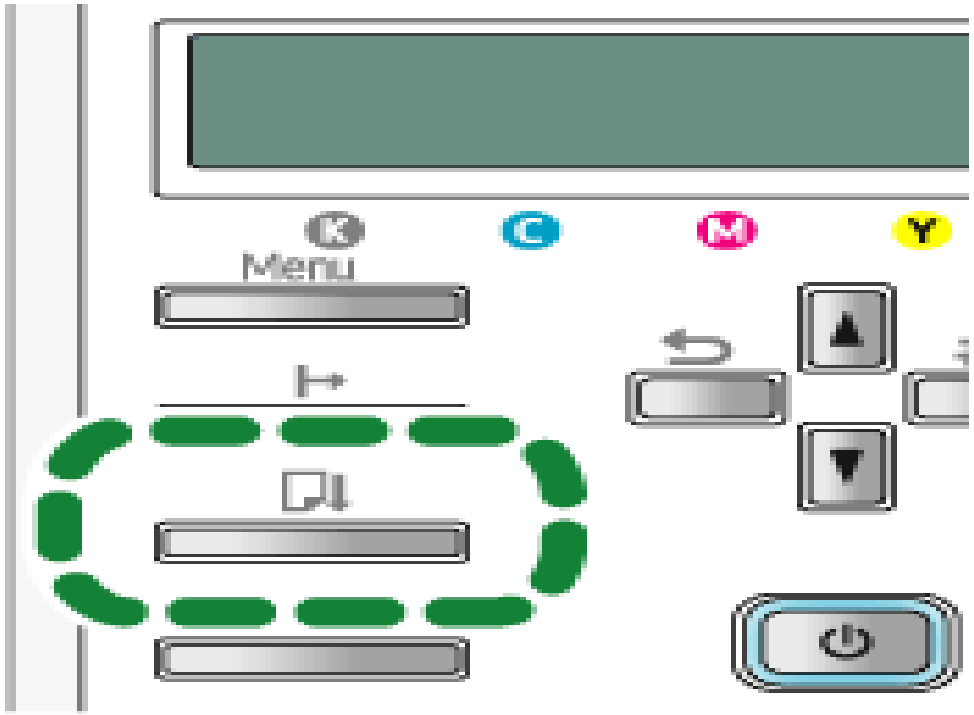
```
public class Main {
  public static void main(String[] args) {
    String txt = "Hello\nWorld!";
    System.out.println(txt);
  }
}
```

## \f form feed

**Form feed is a page-breaking ASCII control character. It directs the printer to eject the current page and to continue printing at the top of another.**



CHU052




BYJ143S

# Java Math

The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

Math.max(x,y)
The Math.max(x,y) method can be used to find the highest value of x and y:

Example
Math.max(5, 10);

```
public class Main {
  public static void main(String[]
args) {
    System.out.println(Math.max(5,
10));
  }
}
```

Math.min(x,y)
The Math.min(x,y) method can be used to find the lowest value of x and y:

Example
Math.min(5, 10);

```
public class Main {
  public static void main(String[]
args) {
    System.out.println(Math.min(5,
10));
  }
}
```

## Math.sqrt(x)

The Math.sqrt(x) method returns the square root of x:

Example
Math.sqrt(64);

```
public class Main {
  public static void main(String[] args) {
    System.out.println(Math.sqrt(64));
  }
}
```

## Math.abs(x)

The Math.abs(x) method returns the absolute (positive) value of x:

Example
Math.abs(-4.7);

```
public class Main {
  public static void main(String[] args) {
    System.out.println(Math.abs(-4.7));
  }
}
```

## Random Numbers

Math.random() returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

Example
Math.random();

```
public class Main {
  public static void main(String[] args) {
    System.out.println(Math.random());
  }
}
```

**To get more control over the random number, for example, if you only want a random number between 0 and 100, you can use the following formula:**

**Example**

```
int randomNum = (int)(Math.random() * 101);  // 0 to 100
```

```
public class Main {
  public static void main(String[] args) {
    int randomNum = (int)(Math.random() * 101);  // 0 to 100
    System.out.println(randomNum);
  }
}
```

## Java Booleans

Very often, in programming, you will need a data type that can only have one of two values, like:

YES / NO
ON / OFF
TRUE / FALSE
For this, Java has a boolean data type, which can store true or false values.

## Boolean Values

A boolean type is declared with the boolean keyword and can only take the values true or false:

Example
```
boolean isJavaFun = true;
boolean isFishTasty = false;
System.out.println(isJavaFun);     // Outputs true
System.out.println(isFishTasty);   // Outputs false
```

```
public class Main {
  public static void main(String[] args) {
    boolean isJavaFun = true;
    boolean isFishTasty = false;
    System.out.println(isJavaFun);
    System.out.println(isFishTasty);
  }
}
```

## Boolean Expression

A Boolean expression returns a boolean value: true or false.

This is useful to build logic, and find answers.

For example, you can use a comparison operator, such as the greater than (>) operator, to find out if an expression (or a variable) is true or false:

Example
```
int x = 10;
int y = 9;
System.out.println(x > y); // returns true,
because 10 is higher than 9
```

```
public class Main {
  public static void main(String[] args) {
    int x = 10;
    int y = 9;
    System.out.println(x > y); // returns true, because 10 is
higher than 9
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    System.out.println(10 > 9); // returns true, because
10 is higher than 9
  }
}
```

**In the examples below, we use the equal to (==) operator to evaluate an expression:**

**Example**

```
int x = 10;
System.out.println(x == 10); // returns true,
because the value of x
```

```
public class Main {
  public static void main(String[] args) {
    int x = 10;
    System.out.println(x == 10); // returns true, because the value of x is equal to 10
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    System.out.println(15 == 10); // returns false, because 10 is not equal to 15
  }
}
```

**Real Life Example**

**Let's think of a "real life example" where we need to find out if a person is old enough to vote.**

**In the example below, we use the >= comparison operator to find out if the age (25) is greater than OR equal to the voting age limit, which is set to 18:**

**Example**

```
int myAge = 25;
int votingAge = 18;
System.out.println(myAge >= votingAge);
```

```
public class Main {
  public static void main(String[] args) {
    int myAge = 25;
    int votingAge = 18;
    System.out.println(myAge >= votingAge); //
returns true (25 year olds are allowed to vote!)
  }
}
```

**Example**
**Output "Old enough to vote!" if myAge is greater than or equal to 18. Otherwise output "Not old enough to vote.":**

```
int myAge = 25;
int votingAge = 18;

if (myAge >= votingAge) {
  System.out.println("Old enough to vote!");
} else {
  System.out.println("Not old enough to vote.");
}
```

```
public class Main {
  public static void main(String[] args) {
    int myAge = 25;
    int votingAge = 18;

    if (myAge >= votingAge) {
      System.out.println("Old enough to vote!");
    } else {
      System.out.println("Not old enough to vote.");
    }
  }
}
```

**Booleans are the basis for all Java comparisons and conditions.**

# Java If ... Else

Java Conditions and If Statements
You already know that Java supports the usual logical conditions from mathematics:

Less than: a < b
Less than or equal to: a <= b
Greater than: a > b
Greater than or equal to: a >= b
Equal to a == b
Not Equal to: a != b

You can use these conditions to perform different actions for different decisions.

## Java has the following conditional statements:

Use **if** to specify a block of code to be executed, if a specified condition is true

Use **else** to specify a block of code to be executed, if the same condition is false

Use **else if** to specify a new condition to test, if the first condition is false

Use **switch** to specify many alternative blocks of code to be executed

## The if Statement

**Use the if statement to specify a block of Java code to be executed if a condition is true.**

**Syntax**

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

**Note that if is in lowercase letters. Uppercase letters (If or IF) will generate an error.**

**In the example below, we test two values to find out if 20 is greater than 18. If the condition is true, print some text:**

**Example**

```
if (20 > 18) {
  System.out.println("20 is greater than 18");
}
```

```
public class Main {
  public static void main(String[] args) {
    if (20 > 18) {
      System.out.println("20 is greater than 18"); // obviously
    }
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    int x = 20;
    int y = 18;
    if (x > y) {
      System.out.println("x is greater than y");
    }
  }
}
```

# The else Statement

**Use the else statement to specify a block of code to be executed if the condition is false.**

**Syntax**

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

**Example**

```
int time = 20;
if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good evening.");
}
// Outputs "Good evening."
```

```
public class Main {
  public static void main(String[] args) {
    int time = 20;
    if (time < 18) {
      System.out.println("Good day.");
    } else {
      System.out.println("Good evening.");
    }
  }
}
```

## else if Statement

**Use the else if statement to specify a new condition if the first condition is false.**

**Syntax**

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

**Example**

```
int time = 22;
if (time < 10) {
  System.out.println("Good
morning.");
} else if (time < 18) {
  System.out.println("Good day.");
} else {
  System.out.println("Good
evening.");
}
// Outputs "Good evening."
```

```
public class Main {
  public static void main(String[] args) {
    int time = 22;
    if (time < 10) {
      System.out.println("Good morning.");
    } else if (time < 18) {
      System.out.println("Good day.");
    }  else {
      System.out.println("Good evening.");
    }
  }
}
```

# Java Short Hand If...Else (Ternary Operator)

## Short Hand If...Else

There is also a short-hand if else, which is known as the ternary operator because it consists of three operands.

It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:

**Syntax**

```
variable = (condition) ? expressionTrue :  expressionFalse;
```

```
public class Main {
  public static void main(String[] args) {
    int time = 20;
    if (time < 18) {
      System.out.println("Good day.");
    } else {
      System.out.println("Good evening.");
    }
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    int time = 20;
    String result;
    result = (time < 18) ? "Good day." : "Good evening.";
    System.out.println(result);
  }
}
```

## Java Switch Statements

**Instead of writing many if..else statements, you can use the switch statement.**

**The switch statement selects one of many code blocks to be executed:**

**Syntax**

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

**This is how it works:**

**The switch expression is evaluated once.**
**The value of the expression is compared with the values of each case.**
**If there is a match, the associated block of code is executed.**
**The break and default keywords are optional, and will be described later in this chapter**
**The example below uses the weekday number to calculate the weekday name:**

**Example**

```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

```
public class Main {
  public static void main(String[] args) {
    int day = 4;
    switch (day) {
      case 1:
        System.out.println("Monday");
        break;
      case 2:
        System.out.println("Tuesday");
        break;
      case 3:
        System.out.println("Wednesday");
        break;
      case 4:
        System.out.println("Thursday");
        break;
      case 5:
        System.out.println("Friday");
        break;
      case 6:
        System.out.println("Saturday");
        break;
      case 7:
        System.out.println("Sunday");
        break;
    }
  }
}
```

## The break Keyword

When Java reaches a break keyword, it breaks out of the switch block.

This will stop the execution of more code and case testing inside the block.

When a match is found, and the job is done, it's time for a break. There is no need for more testing.

A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block.

Example

```
int day = 4;
switch (day) {
  case 6:
    System.out.println("Today is Saturday");
    break;
  case 7:
    System.out.println("Today is Sunday");
    break;
  default:
    System.out.println("Looking forward to the
Weekend");
}
// Outputs "Looking forward to the Weekend"
```

## The default Keyword

The default keyword specifies some code to run if there is no case match:

```
public class Main {
  public static void main(String[] args) {
    int day = 4;
    switch (day) {
      case 6:
        System.out.println("Today is Saturday");
        break;
      case 7:
        System.out.println("Today is Sunday");
        break;
      default:
        System.out.println("Looking forward to the Weekend");
    }
  }
}
```

**Note that if the default statement is used as the last statement in a switch block, it does not need a break.**

## Loops

- can execute a block of code as long as a specified condition is reached.

- are handy because they save time, reduce errors, and they make code more readable.

## Java While Loop

The while loop loops through a block of code as long as a specified condition is true:

**Syntax**

```
while (condition) {
  // code block to be executed
}
```

**Example**

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

```
public class Main {
  public static void main(String[] args) {
    int i = 0;
    while (i < 5) {
      System.out.println(i);
      i++;
    }
  }
}
```

**Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!**

## The Do/While Loop

**The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.**

**Syntax**

```
do {
  // code block to be
executed
}
while (condition);
```

**Example**

```
int i = 0;
do {
  System.out.println(i);
  i++;
}
while (i < 5);
```

```
public class Main {
  public static void main(String[] args) {
    int i = 0;
    do {
      System.out.println(i);
      i++;
    }
    while (i < 5);
  }
}
```

**Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!**

# Java For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

**Syntax**

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

**Example1**

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
      System.out.println(i);
    }
  }
}
```

**Example2**

```
for (int i = 0; i <= 10; i = i + 2) {
  System.out.println(i);
}
```

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i <= 10; i = i + 2) {
      System.out.println(i);
    }
  }
}
```

# Nested Loops

It is also possible to place a loop inside another loop. This is called a nested loop.

The "inner loop" will be executed one time for each iteration of the "outer loop":

```
public class Main {
  public static void main(String[] args) {
    // Outer loop.
    for (int i = 1; i <= 2; i++) {
      System.out.println("Outer: " + i); // Executes 2
times

      // Inner loop
      for (int j = 1; j <= 3; j++) {
        System.out.println(" Inner: " + j); // Executes
6 times (2 * 3)
      }
    }
  }
}
```

# Java Break and Continue

## Java Break

- use to "jump out" of a switch statement.
-can also be used to jump out of a loop.

This example stops the loop when i is equal to 4:

Example

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  System.out.println(i);
}
```

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      if (i == 4) {
        break;
      }
      System.out.println(i);
    }
  }
}
```

## Java Continue

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      if (i == 4) {
        continue;
      }
      System.out.println(i);
    }
  }
}
```

## Break and Continue in While Loop

You can also use break and continue in while loops:

### Break Example

```
public class Main {
  public static void main(String[] args) {
    int i = 0;
    while (i < 10) {
      System.out.println(i);
      i++;
      if (i == 4) {
        break;
      }
    }
  }
}
```

### Continue Example

```
public class Main {
  public static void main(String[]
args) {
    int i = 0;
    while (i < 10) {
      if (i == 4) {
        i++;
        continue;
      }
      System.out.println(i);
      i++;
    }
  }
}
```

# Arrays in Java

## Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

**To declare an array, define the variable type with square brackets:**

Variable that holds an array of strings

```
String[] cars;
```

To insert values to it, you can place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

An array of integers

```
int[] myNum = {10, 20, 30, 40};
```

## Access the Elements of an Array

You can access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```

```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    System.out.println(cars[0]);
  }
}
```

**Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.**

## Change an Array Element

To change the value of a specific element, refer to the index number:

Example

```
cars[0] = "Opel";
```

Example

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
System.out.println(cars[0]);
// Now outputs Opel instead of Volvo
```

```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW",
"Ford", "Mazda"};
    cars[0] = "Opel";
    System.out.println(cars[0]);
  }
}
```

## Array Length

To find out how many elements an array has, use the length property:

Example

```
String[] cars = {"Volvo", "BMW",
"Ford", "Mazda"};
System.out.println(cars.length);
// Outputs 4
```

```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW",
"Ford", "Mazda"};
    System.out.println(cars.length);
  }
}
```

## Java Arrays Loop

Loop Through an Array

You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.

The following example outputs all elements in the cars array:

```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    for (int i = 0; i < cars.length; i++) {
      System.out.println(cars[i]);
    }
  }
}
```

## Loop Through an Array with For-Each

There is also a "for-each" loop, which is used exclusively to loop through elements in arrays:

**Syntax**

```
for (type variable : arrayname) {
  ...
}
```

**Example**

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
  System.out.println(i);
}
```

```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    for (String i : cars) {
      System.out.println(i);
    }
  }
}
```

## Java Multi-Dimensional Arrays

**Multidimensional Arrays**

-A multidimensional array is an array of arrays.

Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.

To create a two-dimensional array, add each array within its own set of curly braces:

**Example**

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

System.out.println(myNumbers[1][2]); // Outputs 7
```

```
public class Main {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    System.out.println(myNumbers[1][2]);
  }
}
```

## Change Element Values

You can also change the value of an element:

**Example**

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
myNumbers[1][2] = 9;
System.out.println(myNumbers[1][2]); // Outputs 9 instead of 7
```

```
public class Main {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    myNumbers[1][2] = 9;
    System.out.println(myNumbers[1][2]); //
Outputs 9 instead of 7
  }
}
```

## Loop Through a Multi-Dimensional Array

**We can also use a for loop inside another for loop to get the elements of a two-dimensional array (we still have to point to the two indexes):**

**Example**

```
public class Main {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    for (int i = 0; i < myNumbers.length; ++i) {
      for(int j = 0; j < myNumbers[i].length; ++j) {
        System.out.println(myNumbers[i][j]);
      }
    }
  }
}
```

```
public class Print2DArray {
   public static void main(String[] args) {
      final int[][] matrix = {
         { 1, 2, 3 },
         { 4, 5, 6 },
         { 7, 8, 9 }
      };
      for (int i = 0; i < matrix.length; i++) { //this equals to the row in our matrix.
         for (int j = 0; j < matrix[i].length; j++) { //this equals to the column in each row.
            System.out.print(matrix[i][j] + " ");
         }
         System.out.println(); //change line on console as row comes to end in the matrix.
      }
   }
}
```