# Core Java

A general-purpose programming language that is class-based, object-oriented, and designed to have as few implementation dependencies as possible.


Cognixia™

# Outline

1. Introduction
2. Basic Programming
3. Classes & objects
4. Packages
5. Dates
6. Flow Control
7. Enums
8. Strings
9. Regular Expressions
10. Arrays
11. Inheritance & Composition
12. Interfaces

# Prerequisites


1. JDK Version 8
   - Select the **executable version** for your OS
   - Once downloaded, if on Windows, make sure **java is on your path**: How to set up Path
2. Eclipse Download
   - Select option for **"Eclipse IDE for Enterprise Java Developers"**
   - Create a folder in root directory called **"Java_Workspace"** and set your workspace to this folder when asked
   - Once opened, click **Help -> Eclipse Marketplace -> Search for "Spring Tools 4" -> Install**

# Introduction to Java

# What is Java?

- Java is a **programming language** and a platform
- **Platform** – any hardware r software environment in which a program runs
- Used in 9 billion devices around the world
- Used in 4 types of Applications:
  - Standalone Application
  - Web Application
  - Enterprise Application
  - Mobile Application

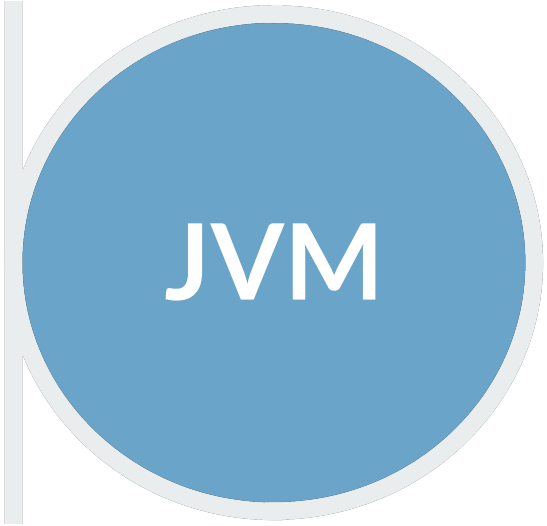
# Why Java?

According to Sun, the **Java** language is simple because:

- ➔ **Simple syntax**, based in C++ (easier to learn it after C++)
- ➔ Removed confusing and/or rarely-used features (explicit pointer, operator overloading etc.)
- ➔ No need to remove unreferenced objects, there is **Automatic Garbage Collection**
- ➔ **Object-oriented**
- ➔ **Architecture neutral**

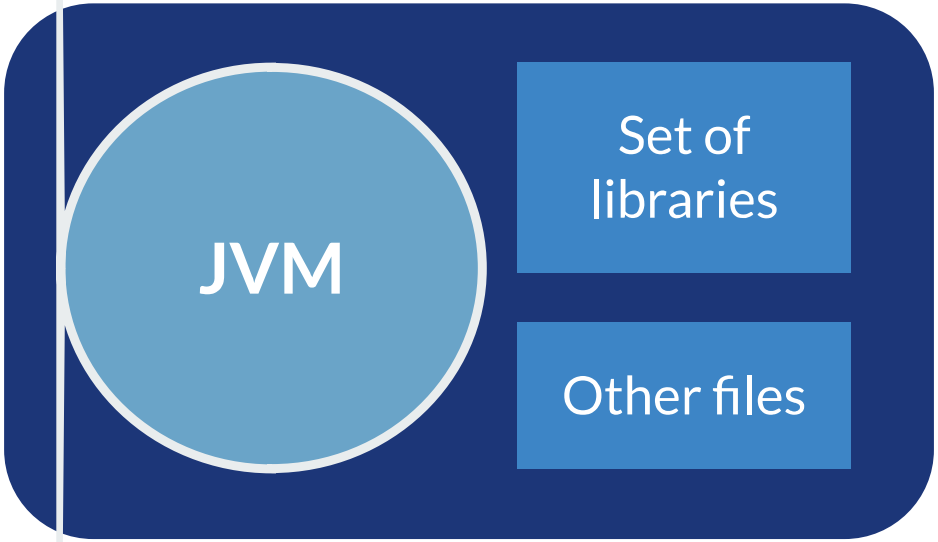
# Java Virtual Machine (JVM)

- Abstract machine
- **Provides runtime environment** for java bytecode to be executed
- Main tasks:
  - *Loads code*
  - *Verifies code*
  - *Executes code*
  - *Provides runtime environment*



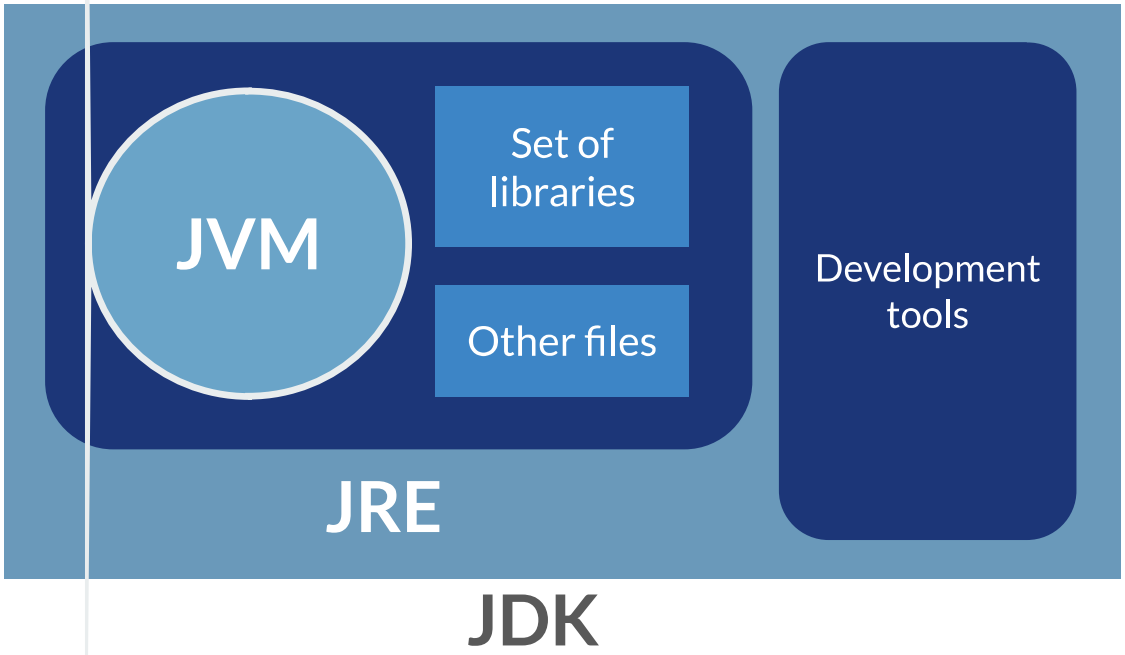
# Java Runtime Environment (JRE)

- ➔ Provides runtime environment
- ➔ **Implementation of JVM**
- ➔ Physically exists
- ➔ **Contains libraries + other files that JVM uses at runtime**

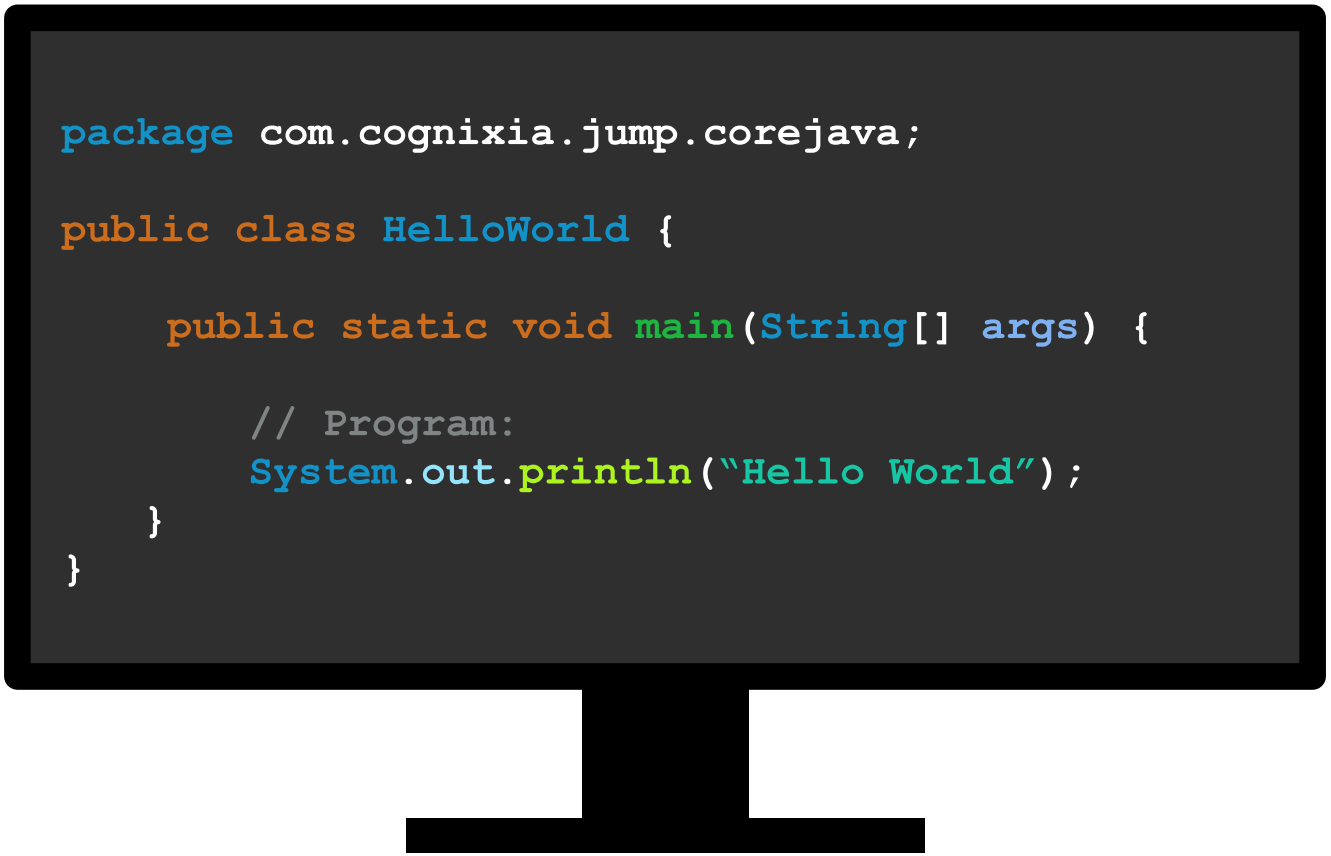


# Java Development Kit (JDK)

- ➔ Physically exists
- ➔ **Contains JRE + development tools**



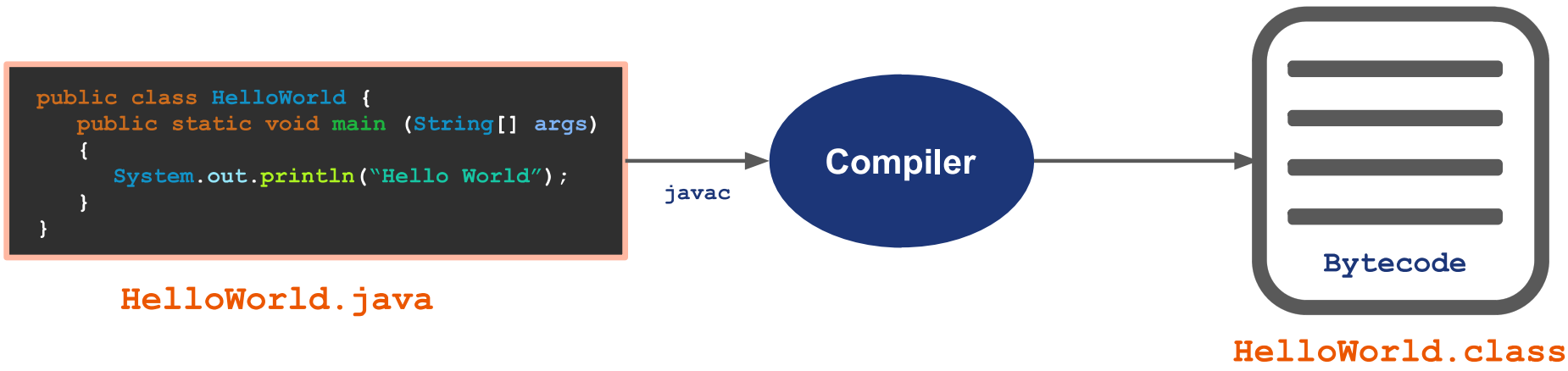
# Hello World: First Program



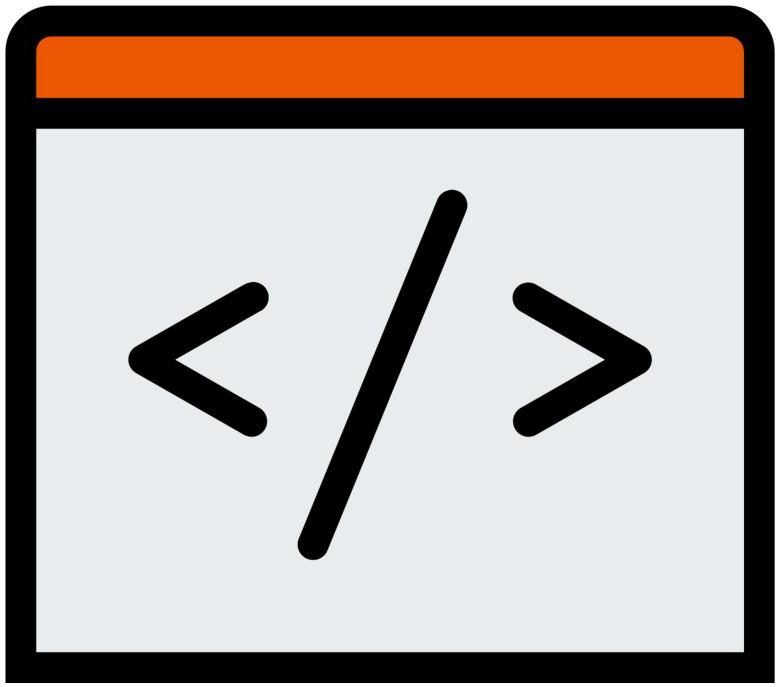
```java
package com.cognixia.jump.corejava;

public class HelloWorld {

    public static void main(String[] args) {

        // Program:
        System.out.println("Hello World");
    }
}
```

# What Happens at Compile Time?

At **compile time**, the java file is compiled by the Java Compiler (does not interact with the OS) and converts the java code into bytecode.



# Basic Programming

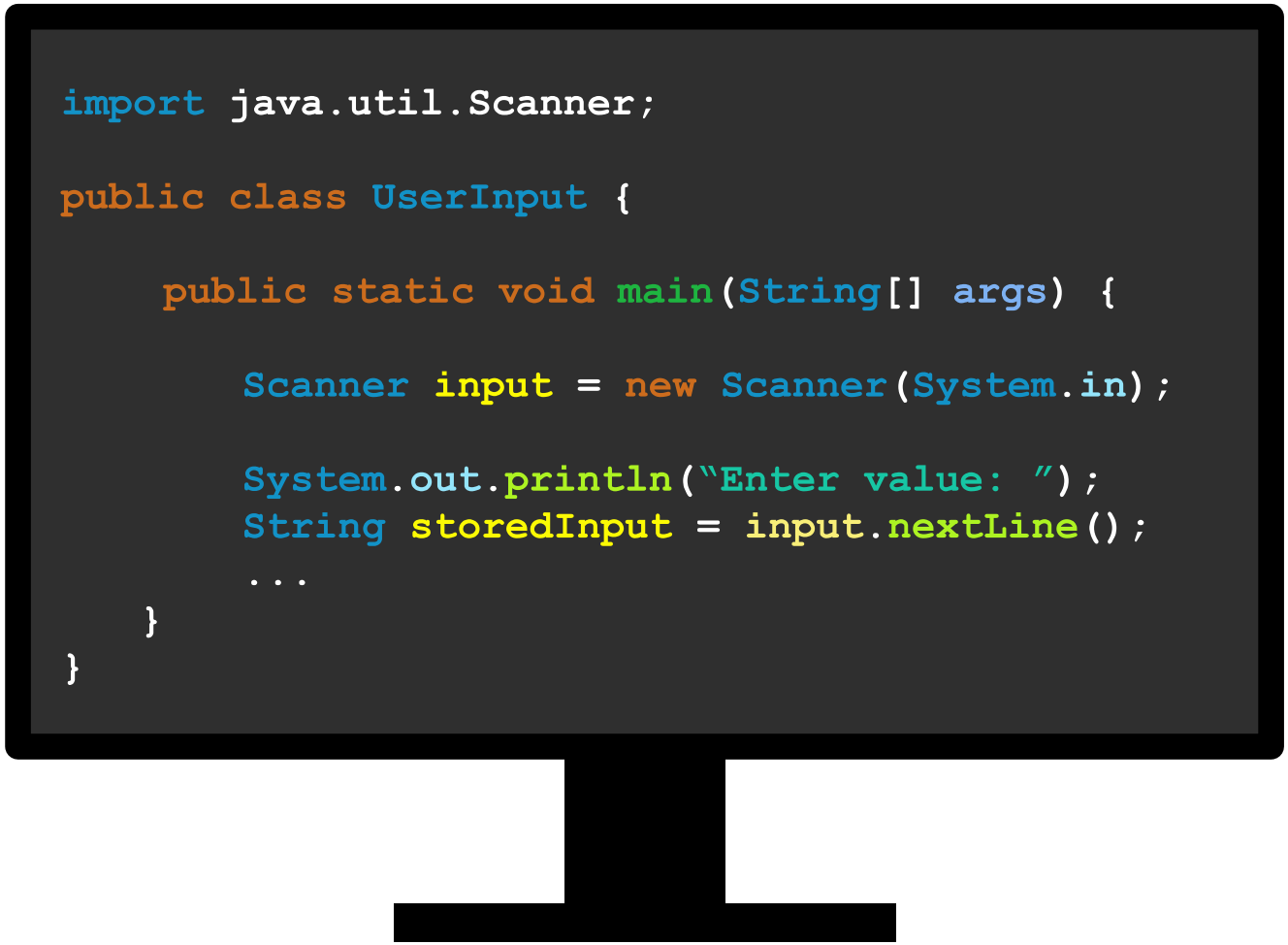


# Variable Types

**Primitives** - most basic data type in Java, hold pure, simple values of a kind

| Name | byte | short | int | long | float | double | char | boolean |
|---|---|---|---|---|---|---|---|---|
| Value | number | number | number | number | float number | float number | character | true or false |
| Size | 1 byte | 2 byte | 4 byte | 8 byte | 4 byte | 8 byte | 2 byte | 1 byte |

# Read From the Console



# Final Keyword

**Final** - a constant in Java

Final can apply to:

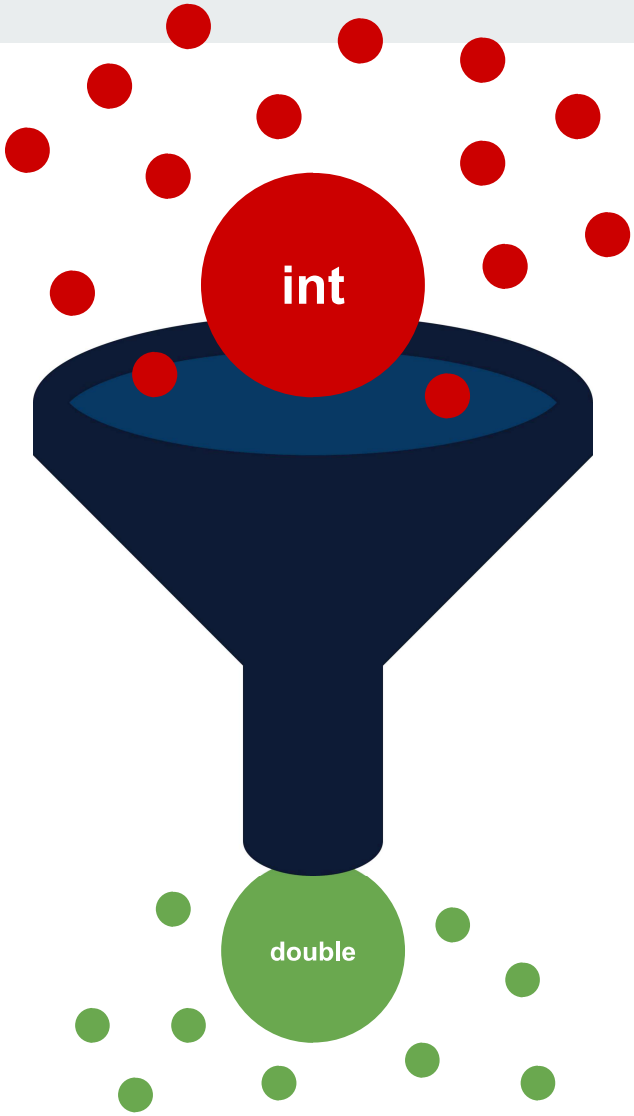
- **Variables**
  - Value cannot be changed.
- **Methods**
  - Method cannot be overridde
- **Classes**
  - Class cannot be inherited

# Casting Variables

**Casting** variables is explicitly convert one type to another.

**Primitives** and **Objects** can be converted between one another using casting.

```
// converts from double to int
double dubs = 5.0;
int num = (int) dubs;
```



# Order of Precedence (High to Low)

**Order of Precedence** - order in which operators in an expression are evaluated

**Oracle Documentation:**

https://docs.oracle.com/cd/E19253-01/817-6223/chp-typeopexpr-12/index.html

| | Important to note |
|---|---|
| 1 | `!, +, - (unary)` |
| 2 | `*, /, %` |
| 3 | `+, -` |
| 4 | `<, <=, >=, >` |
| 5 | `==, !=` |
| 6 | `&&` |
| 7 | `\|\|` |
| 8 | `= (assignment)` |

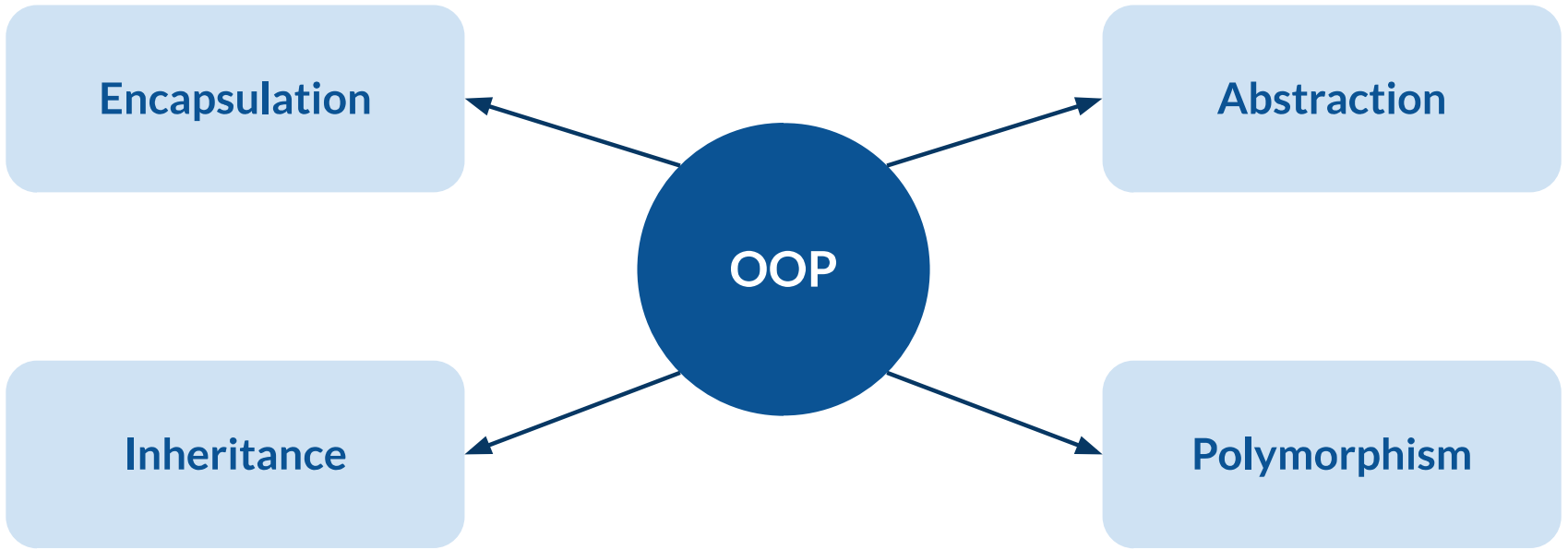
# Primitive v Autoboxed

- Java is NOT a pure OOP language
  - has primitives
- Objects are required in many applications for Java
- Wrapper Classes - objects that wrap around the primitive type
- Can use Collections - Collections cannot store primitives on their own

```
double dubs = 5.0;
int num = (int) dubs;

// passes int to boxed type
Integer boxed = num;
```

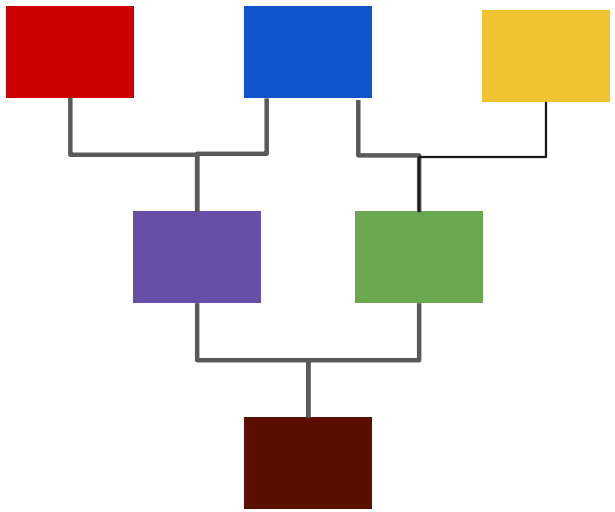

JAVA AEROBICS

# Object Oriented Programming (OOP)

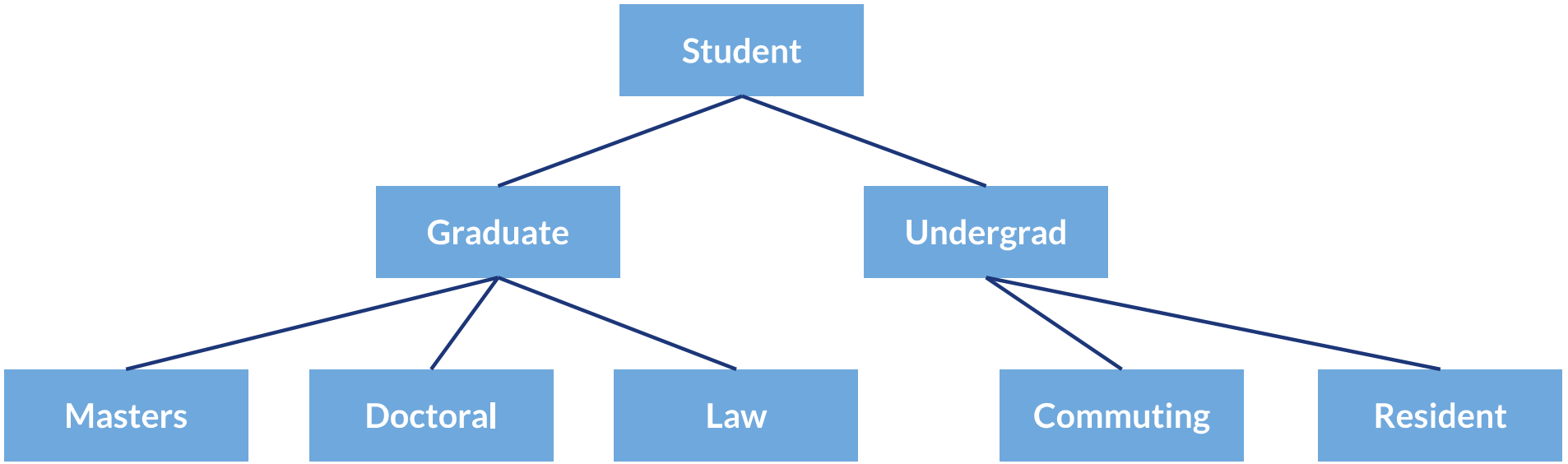


# OOP - Inheritance

- ➔ ***Inheritance*** is a mechanism in OOP to design two or more entities that are different but share many common features
- ➔ Features common to all classes are defined in the ***superclass***
- ➔ The classes that inherit common features from the superclass are called ***subclasses***
- ➔ We also call the superclass an ***ancestor*** and the subclass a ***descendant***

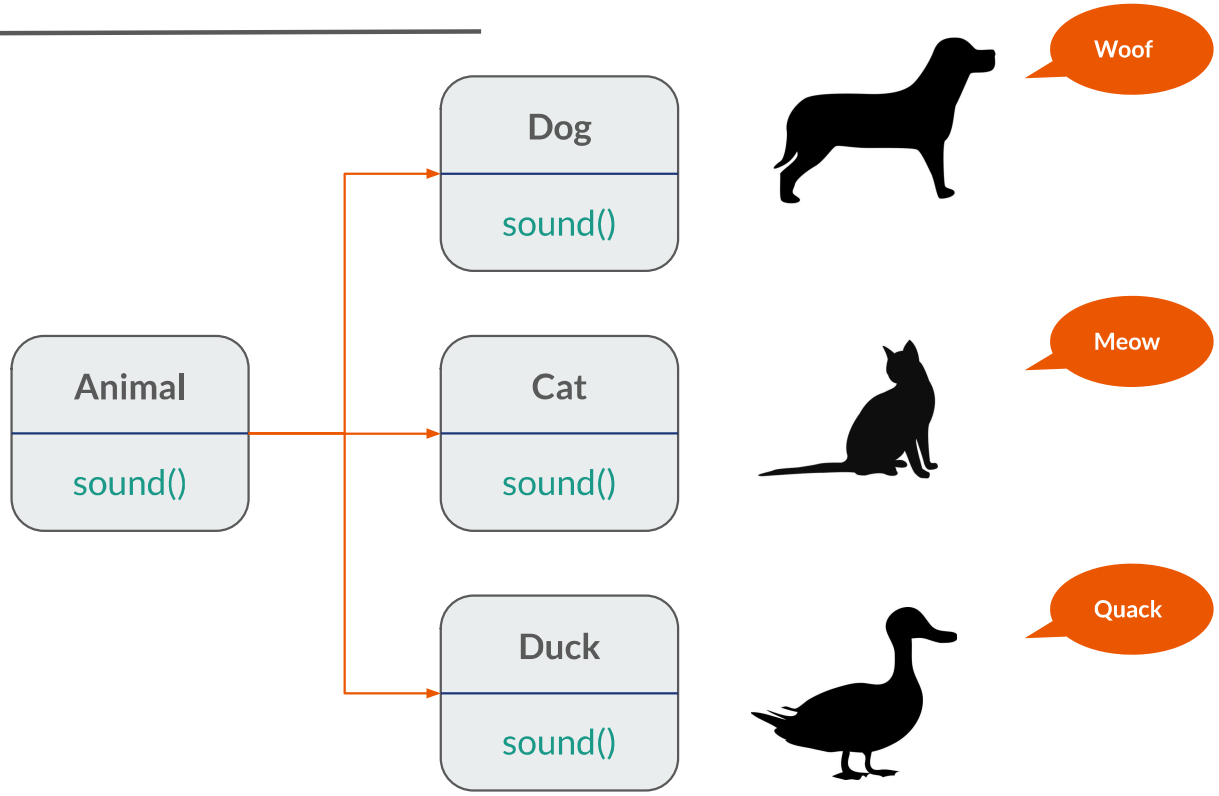


# OOP - Inheritance



# OOP - Polymorphism

- ***Polymorphism*** is a mechanism in OOP where one element of code can have many forms
  - Polymorphism can be implemented in such as **objects** and **classes**, and **class methods**



# OOP - Abstraction and Encapsulation

**Abstraction** as a concept of OOP enforces "***data hiding***". That is, only relevant code is displayed, so that code is layered.

**Encapsulation** is a "***data grouping***". Think of this as a protective shield around code. An example would be grouping functions together in class.

# WHITE BOARD EXERCISE

# Creating a Class Diagram

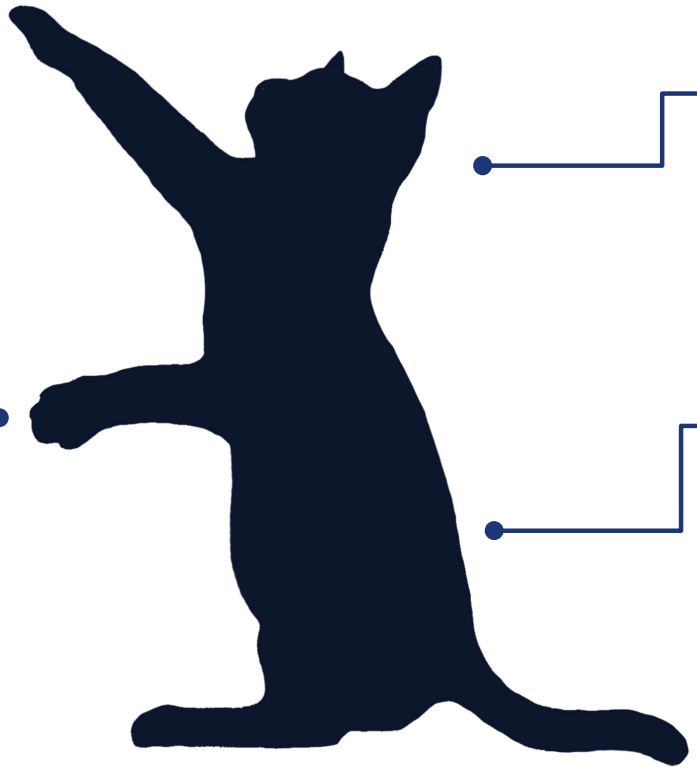
| Create Class | Class Properties | Child Class | Polymorphism | Explain |
| --- | --- | --- | --- | --- |
| Choose a topic and create a class for this, draw it up on the board | Create attributes and methods for this class. | Create a child class that can inherit from this original class. Come up with attributes and methods for this child class. | Create a method that will override one of the methods from the parent. | What is happening in this diagram? Is there encapsulation? |

# Classes & Objects

*Blueprint*

*Houses built according to the blueprint*

# Classes



| Class |
| --- |
| Cat |

| Methods |
| --- |
| Meow |
| Play |
| Sleep |

| Attributes |
| --- |
| Name |
| Favorite Toy |

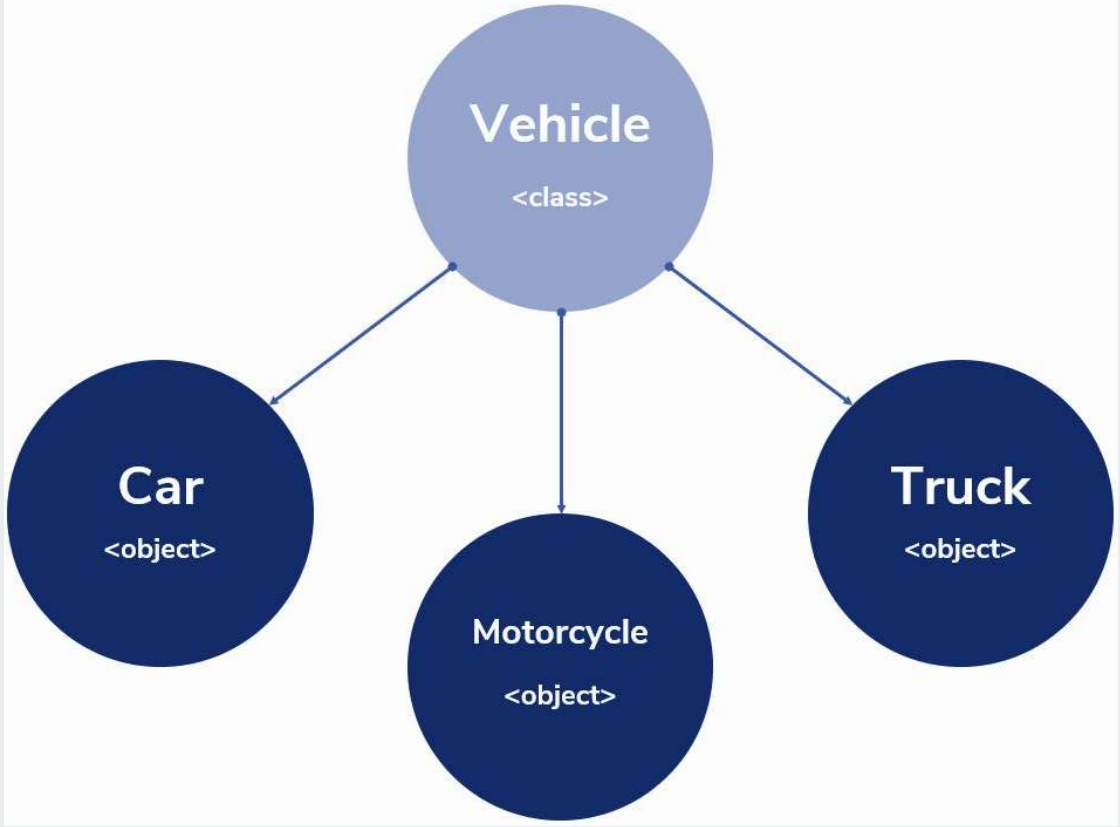
# Classes and Objects

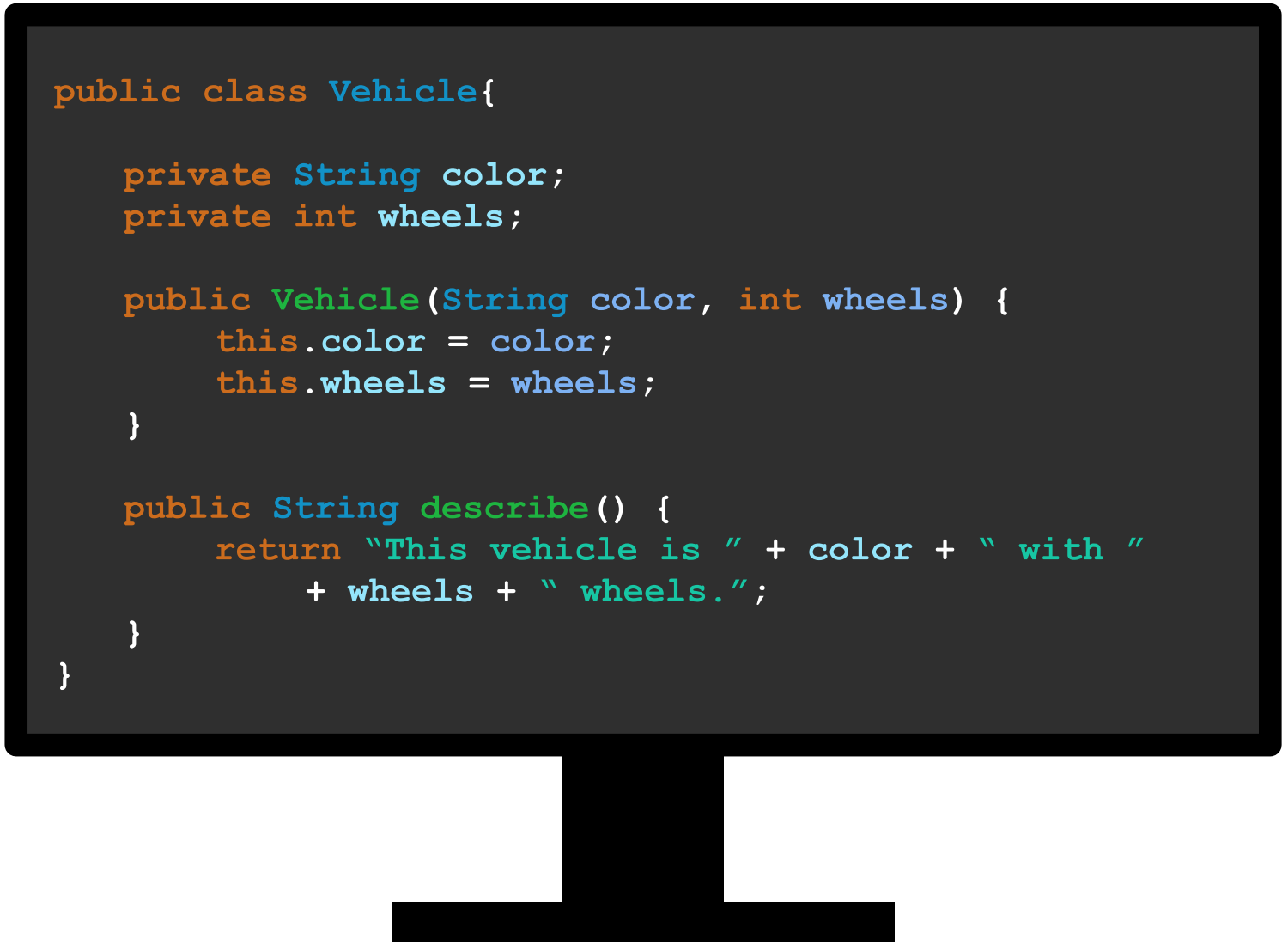
Object-oriented programs use **objects**.

An ***object*** is a thing, both tangible and intangible.

To create an object inside the computer program, we must provide a definition for objects—how they behave and what kinds of information they maintain —called a ***class***.

An object is called an ***instance*** of a class.


Vehicle
<class>
Car
<object>
Motorcycle
<object>
Truck
<object>


public class Vehicle{
private String color;
private int wheels;
public Vehicle(String color, int wheels) {
this.color = color;
this.wheels = wheels;
}
public String describe() {
return "This vehicle is " + color + " with "
+ wheels + " wheels.";
}
}

# Messages and Methods

To instruct a class or an object to perform a task, we send a ***message*** to it.

You can send a message only to the classes and objects that understand the message you sent to them.

A class or an object must possess a matching ***method*** to be able to handle the received message.

# Messages and Methods

A method defined for a class is called a ***class method***, and a method defined for an object is called an ***instance method***.

A value we pass to an object when sending a message is called an ***argument*** of the message.

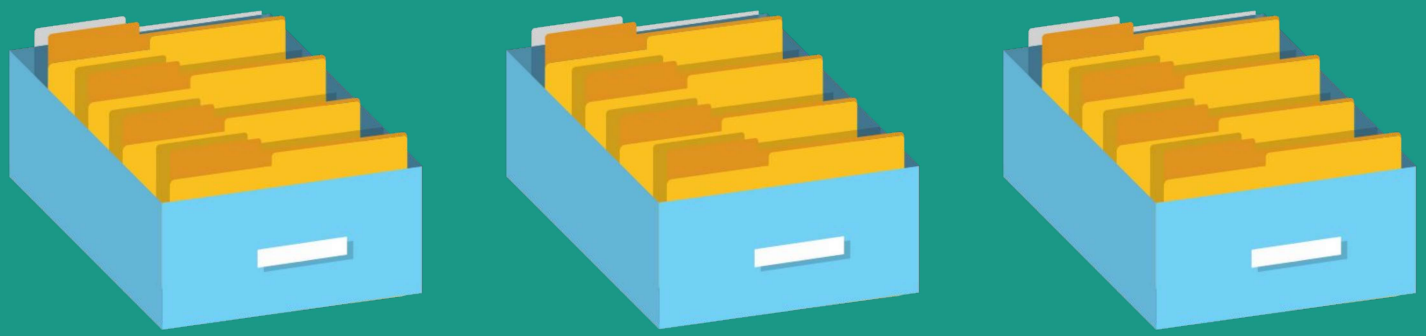
# Access Modifiers

| Modifier | Class | Package | Subclass | Global |
|---|---|---|---|---|
| Public | Allowed | Allowed | Allowed | Allowed |
| Protected | Allowed | Allowed | Allowed | Denied |
| Default | Allowed | Allowed | Denied | Denied |
| Private | Allowed | Denied | Denied | Denied |

# Class Example

```
public class Animal {

    // attributes here

    // create constructor

    // define methods

}
```

# Packages and Imports



# Packages and Imports

- **Packages** - Java mechanism to organize classes
  - Naming Conventions:
    - i.e. - com.cognixia.jump.corejava
    - OrganizationType.CompanyName.OrganizationTopic
- **Imports** - Java needs to know what libraries to reference

  to use certain Classes
  - I.e. - java.lang, java.util

# Dates


SEP
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

# java.util.date

- Java comes with **Date** package, imported from **java.util**
- New Date object is created, given current date from system
- Date objects contain a toString( ) method, converts date to readable string

```
import java.util.Date;

public class PrintDate {

    public static void main(String[]
        args) {

        Date today = new Date();
        today.toString();
        // Sun Dec 15 16:58:01 EST 2019
    }
}
```

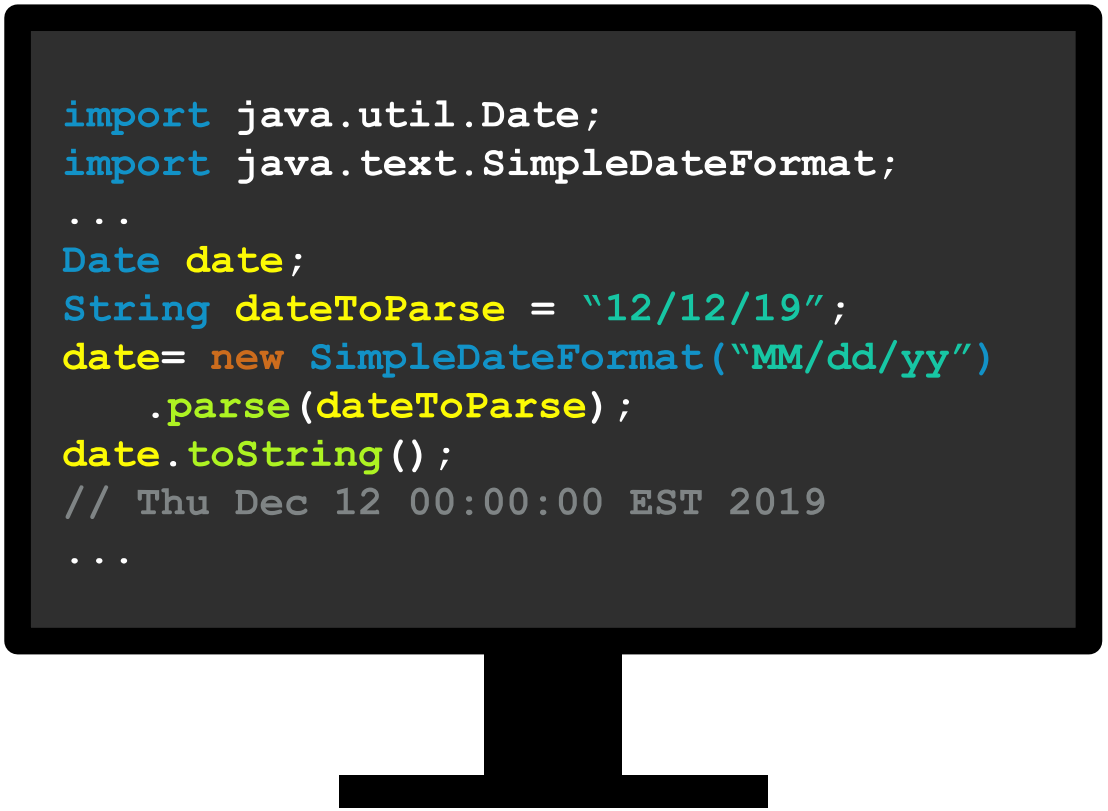
# Formatting Dates

- ➔ **SimpleDateFormat** can be used to change how the date is presented
- ➔ By passing desired format (ex. Month/Day/Year) in the ***SDF*** constructor, ***Date*** object can be displayed with any desired format

```
import java.util.Date;
import java.text.SimpleDateFormat;

...
Date today = new Date();
SimpleDateFormat sdf;
sdf = new SimpleDateFormat("MM/dd/yy");
sdf.format(today);
// 12/15/19
...
```

# String to Date

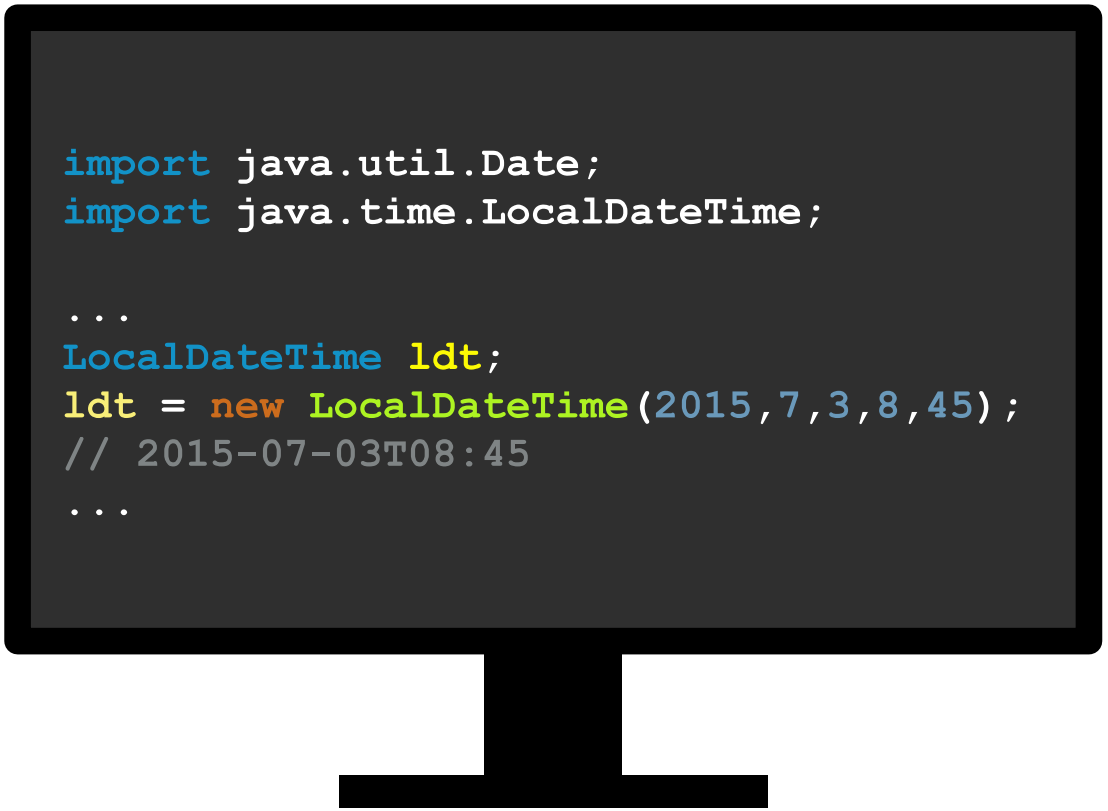
- ➔ **SimpleDateFormat** can also convert date string to Date object
- ➔ The string “12/12/19” is converted to date object
- ➔ SDF object is created to parse incoming string to useable date



```
import java.util.Date;
import java.text.SimpleDateFormat;
...
Date date;
String dateToParse = "12/12/19";
date= new SimpleDateFormat("MM/dd/yy")
    .parse(dateToParse);
date.toString();
// Thu Dec 12 00:00:00 EST 2019
...
```

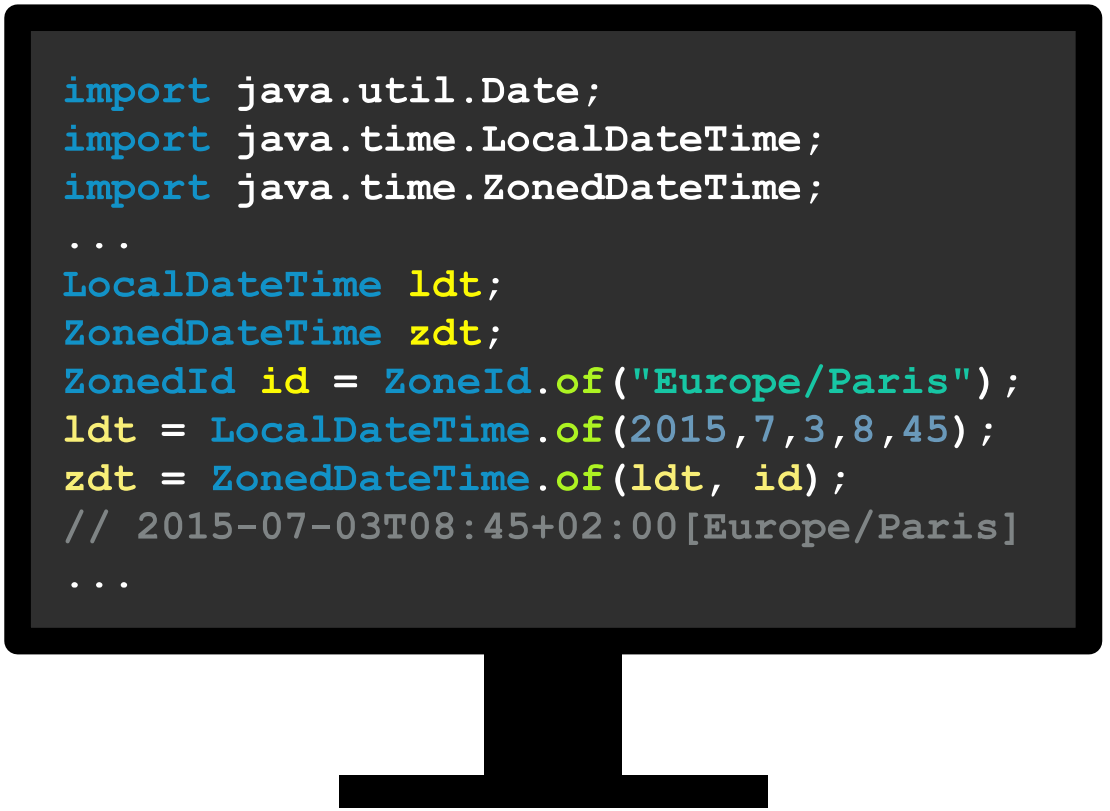
# LocalDateTime

- **LocalDateTime** used to represent a combination of date and time
- By passing in ***year, month, day,hour, and minute***; a new LocalDateTime object is created



# ZonedDateTime

- ➔ **ZonedDateTime** used to create an object representing a date in a given time zone
- ➔ **ZoneId** object created with the desired time zone
- ➔ ZonedDateTime created with the local date and time and the ZoneId.



```
import java.util.Date;
import java.time.LocalDateTime;
import java.time.ZonedDateTime;
...
LocalDateTime ldt;
ZonedDateTime zdt;
ZonedId id = ZoneId.of("Europe/Paris");
ldt = LocalDateTime.of(2015,7,3,8,45);
zdt = ZonedDateTime.of(ldt, id);
// 2015-07-03T08:45+02:00[Europe/Paris]
...
```