



# Core Java

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



---

# Outline



1. Introduction
2. Basic Programming
3. Flow Control
4. Classes & Objects
5. Inheritance & Composition
6. Arrays
7. Strings
8. Regular Expressions
9. Enums
10. Dates
11. Interfaces
12. Packages & Imports
13. Collections and Generics
14. Exceptions & Assertions
15. JUnit

# Prerequisites



1. JDK Version 11
  - If on Windows, make sure **java is on your path**: [How to set up Path](#)
  - At least Java 11
2. Eclipse Download (Windows Users)
  - 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**
3. STS Download (Mac/Linux Users)
  - Click on download for STS on Eclipse

---

# Introduction to Java



# What is Java?

---

- Java is a **programming language** and a platform
- **Platform** – any hardware or 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:

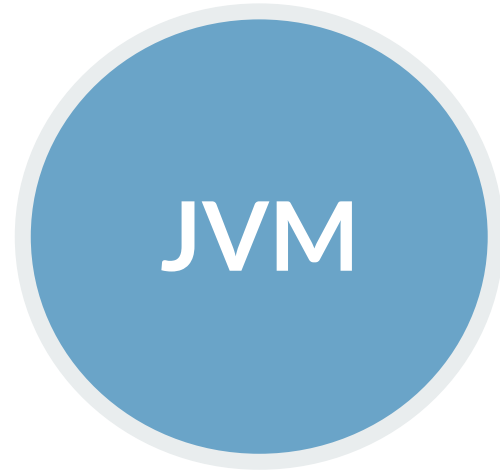
- **Simple syntax**, based in C++ (easier to learn it after C++)
- Removed confusing and/or rarely-used features (explicit pointer, operator overloading etc.)
- No need to remove unreferenced objects, there is **Automatic Garbage Collection**
- **Object-oriented**
- **Architecture neutral**





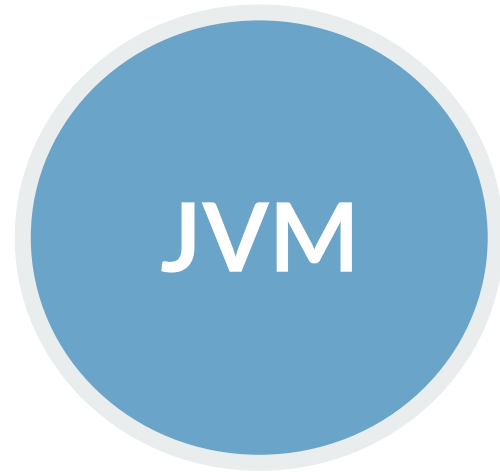
# Java Virtual Machine (JVM)

→ What is a **Virtual Machine**?



# 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*





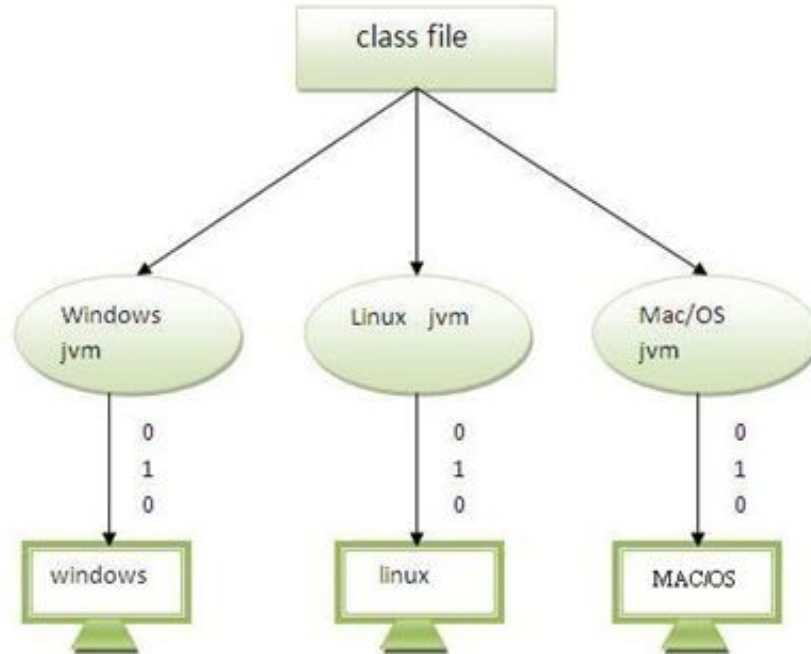
# JVM - Platform Independence

---

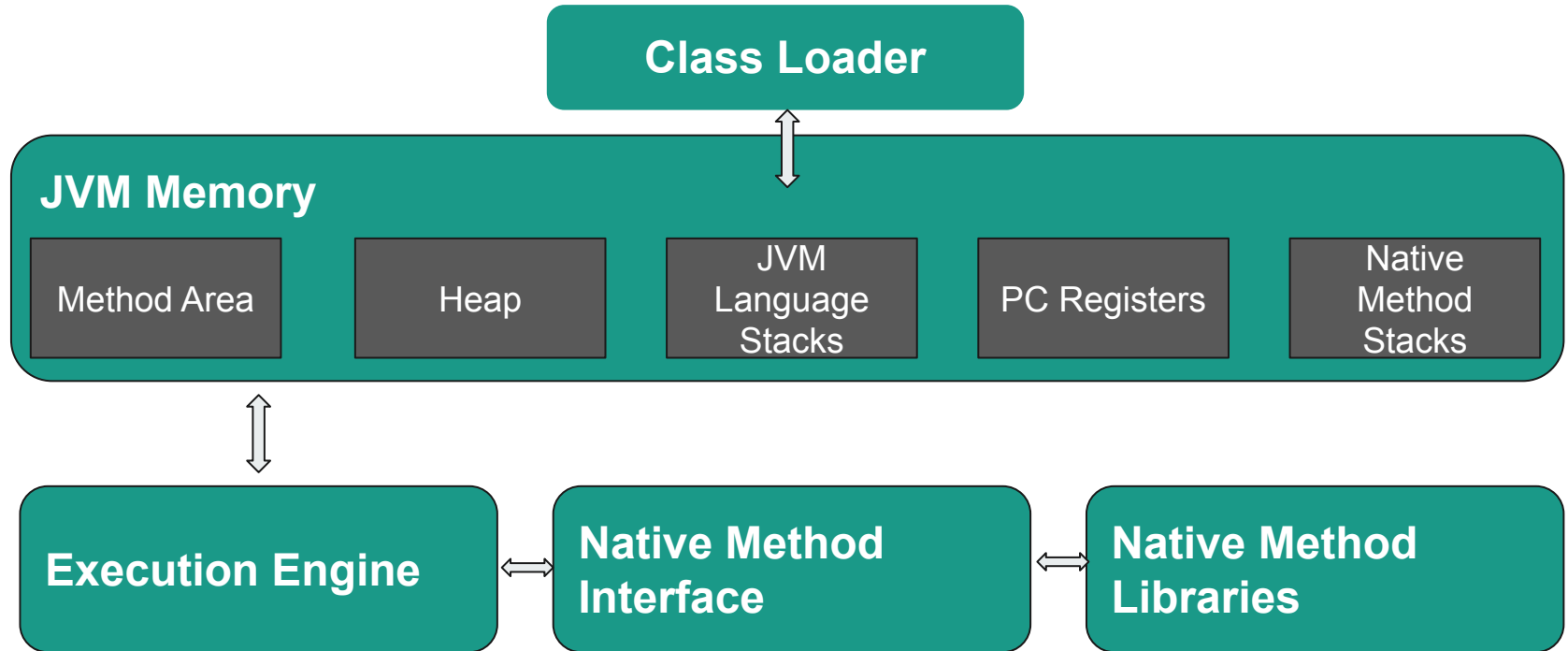
- Java code can be run on **multiple platforms**
  - Windows, Linux, Sun Solaris, Mac/OS
- Java code is compiled by the compiler and converted into **bytecode**
- This **bytecode is a platform independent** code because it can be run on multiple platforms
  - Write Once and Run Anywhere (WORA)

# JVM - Platform Independence

---

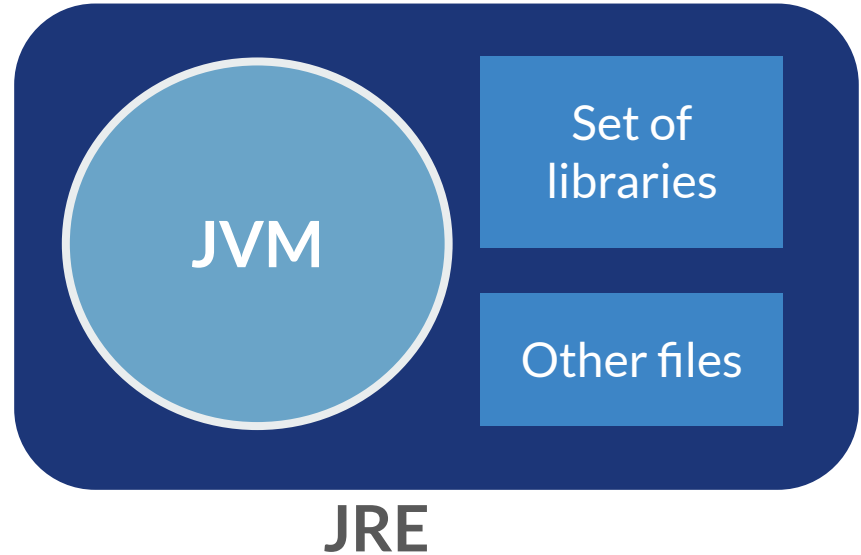


# JVM - Architecture



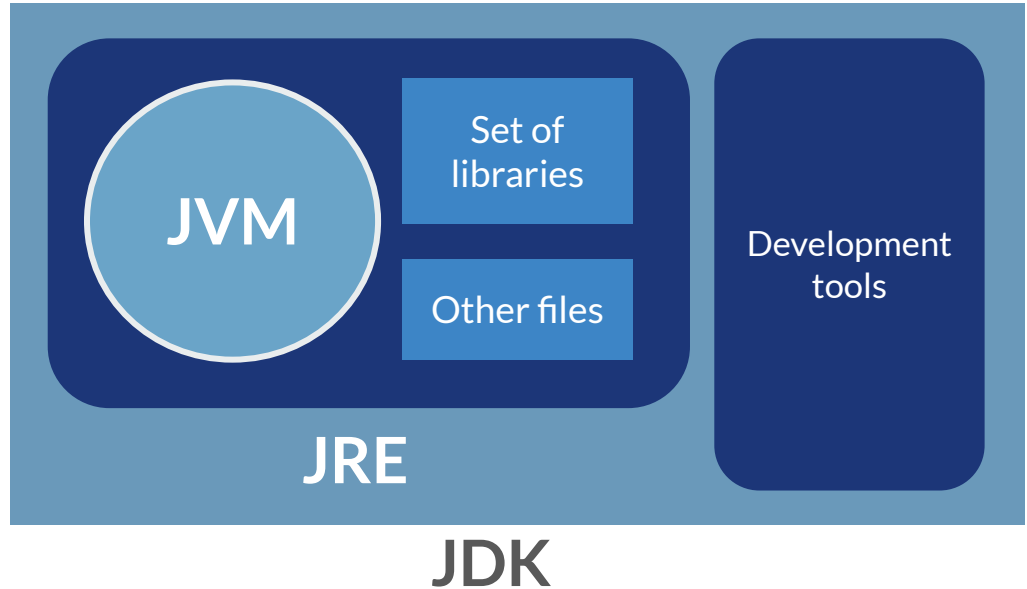
# Java Runtime Environment (JRE)

- 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**



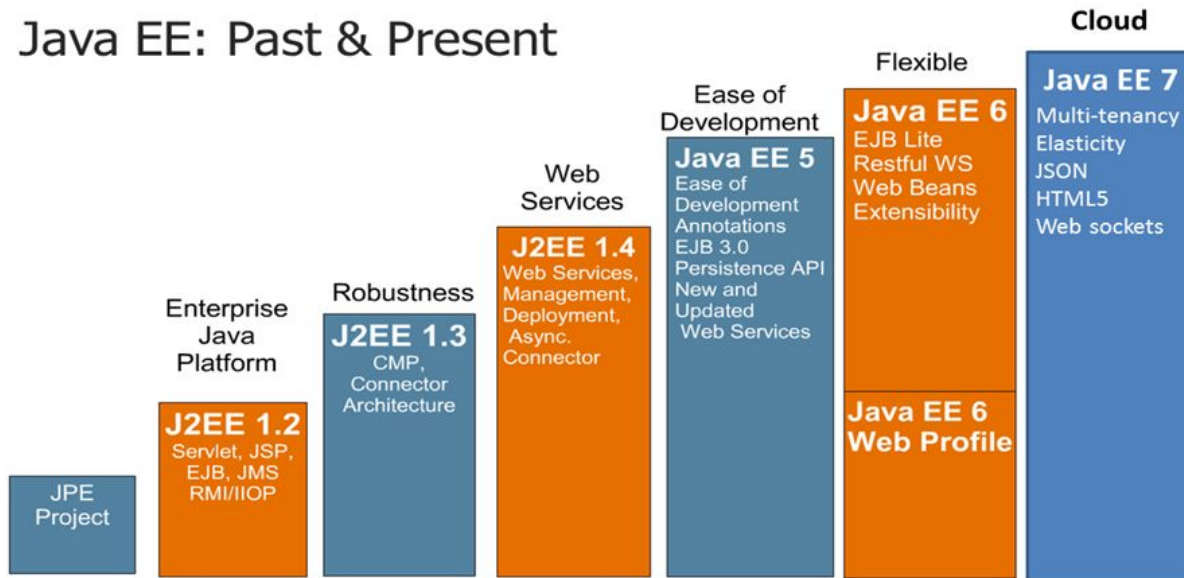


# Java Editions

- **Java Standard Edition (J2SE)**
  - Core Java Platform targeting applications running on workstations
- **Java Enterprise Edition (J2EE / JEE)**
  - Component-based approach to developing distributed, multi-tier enterprise applications
- **Java Micro Edition (J2ME)**
  - Targeted at small, stand-alone or connectable consumer and embedded devices

# Java - Continued Evolution

## Java EE: Past & Present





# Java - JAR , WAR, and EAR

## JAR:

EJB modules which contains enterprise java beans class files and EJB deployment descriptor are packed as JAR files with .jar extension

## WAR:

Web modules which contains Servlet class files, JSP Files, supporting files, GIF and HTML files are packaged as JAR file with .war( web archive) extension

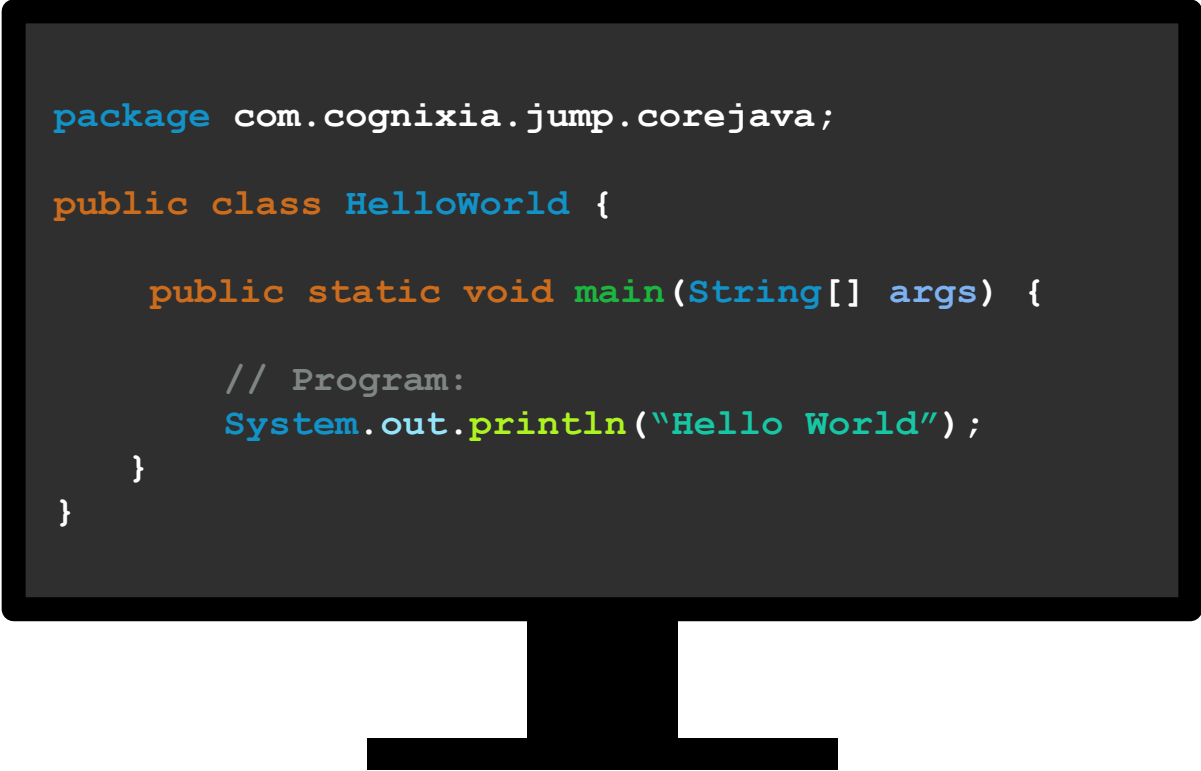
## EAR:

All above files(.jar and .war) are packaged as JAR file with . ear ( enterprise archive) extension and deployed into Application Server



# Hello World: First Program

---



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

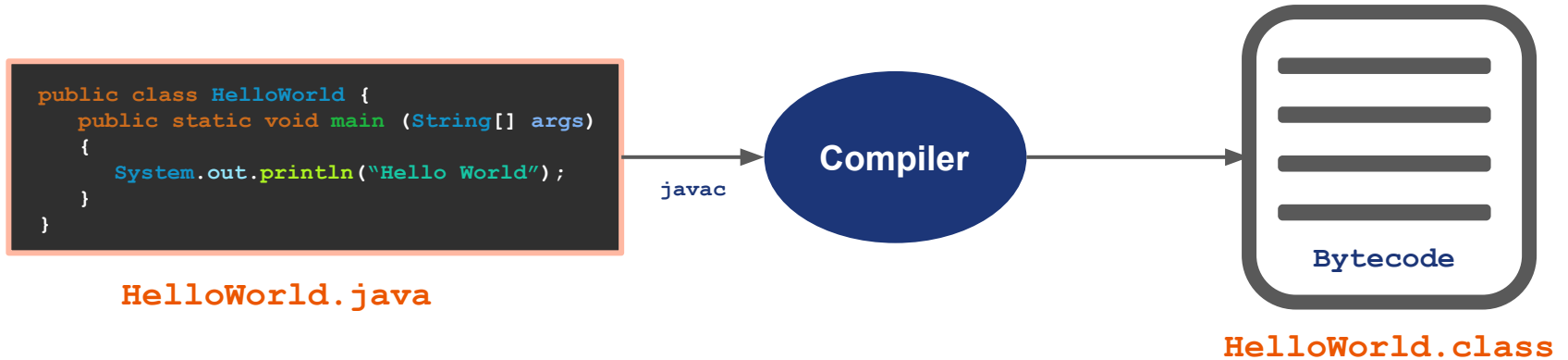
public class HelloWorld {

    public static void main(String[] args) {

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

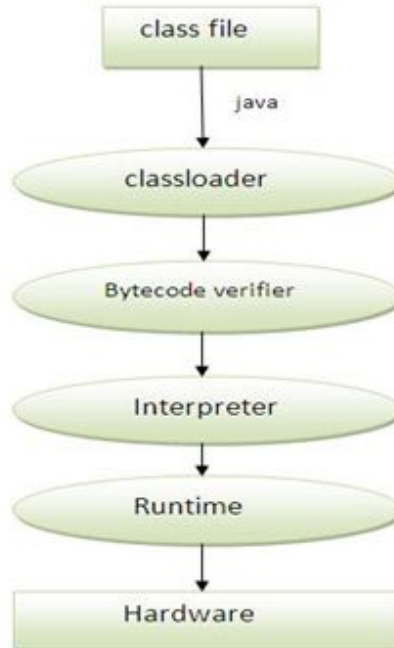
# What Happens at Compile Time?

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.



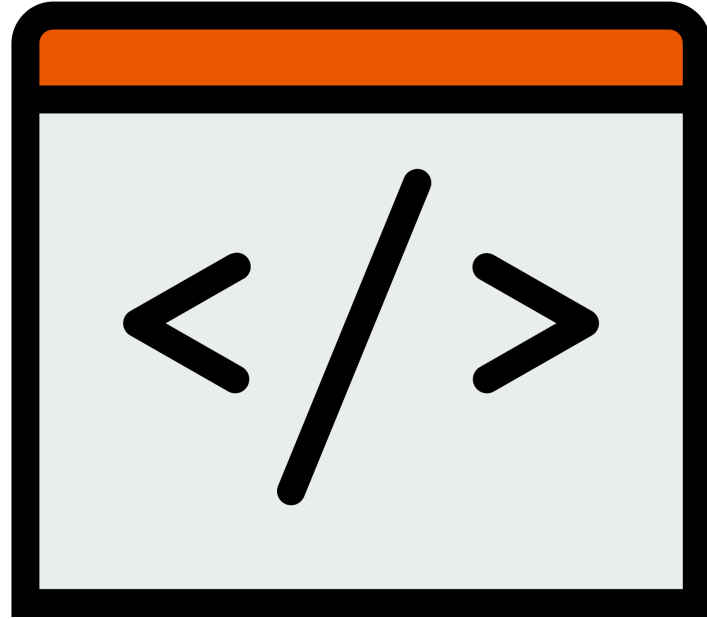
# What Happens at Runtime?

---



---

# 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

# The main method

```
public class UserInput {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

Modifier

public

Modifier

static

Return Type

void

Method Name

main(

Parameter

.....  
String[ ] args ) {

# Final Keyword

**Final** - a constant in Java

Final can apply to:

- **Variables**
- **Methods** and **Classes** (covered later)
- Are Immutable - *constant*



```
final double PI = 3.14159;  
final int MONTH_IN_YEAR = 12;  
final short FARADAY_CONSTANT = 23060;
```

↑  
The reserved word **final** is used to declare constants.

↑  
These are constants, also called *named constant*.

↑  
These are called *literal constant*.

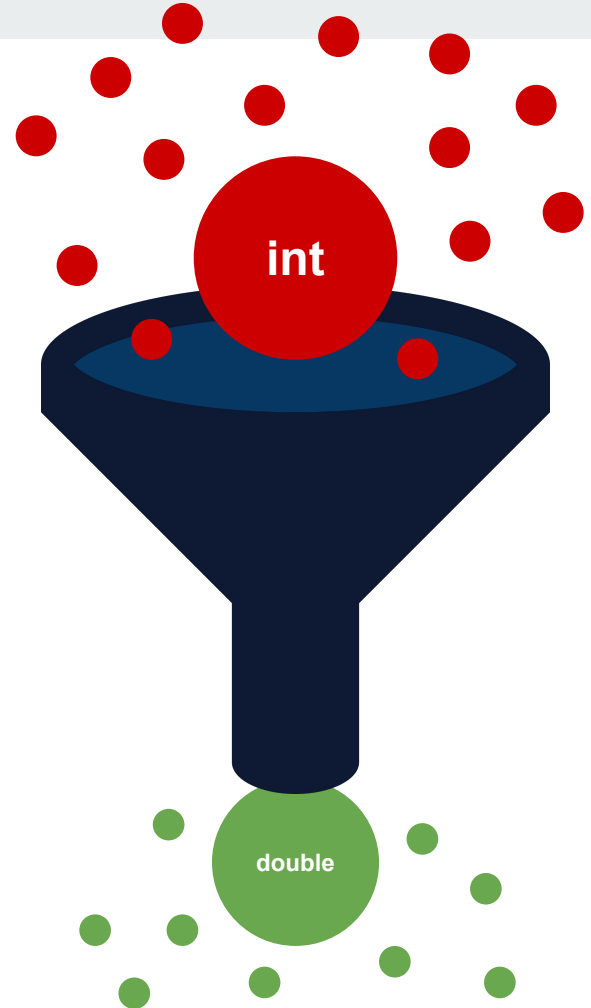
# Casting Variables

---

**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)

# Read From the Console

```
import java.util.Scanner;

public class UserInput {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.println("Enter value: ");
        String storedInput = input.nextLine();
        ...
    }
}
```

# Flow Control



# Conditionals: If/Else

---

- Conditionals are a core part of nearly every programming language.
- Java uses **if / else if / else** syntax to control the flow of a program

```
if (condition1){  
    // this code will execute if condition1 a strict true boolean  
}  
else if (condition2){  
    // this code will execute if condition2 a strict true boolean  
    // and condition1 is a strict false boolean  
}  
else {  
    // this code will execute if neither condition1 or condition2  
    // are strict true booleans  
}
```

# Conditionals: Nested If Statements

---

→ Nesting if statements can test conditions that are reliant on the state of other conditions

```
if (condition1) {  
    if (condition2) {  
        // this code will execute if condition1 and condition2 are  
        // strict true booleans  
    } else {  
        // this code will execute if condition1 is a strict true  
        // boolean, and condition2 is a strict false boolean  
    }  
} else {  
    // this code will execute if condition1 is a strict false boolean,  
    // but has no relationship to condition2  
}
```

# Conditionals: Logical Operators

---

**Logical operators** can be used to check the conditions on primitive data types

- `<` → Less than
- `>` → Greater than
- `<=` → Less than or equal to
- `>=` → Greater than or equal to
- `==` → Equal to (NOTE: This compares memory locations)
- `!=` → Not equal to
- `!` → Not (reverses a boolean)
- `&&` → And (True if Both booleans are true)
- `||` → Or (True if at least one boolean is true)
- `^` → XOR (True if one boolean is true and the other false)

Important note: Strings are objects, not primitives, these operators will not work as properly on Strings.

# Conditionals: Switch

---

- Switch statements are a more compact syntax for conditionals.
- Switch statements can be used to direct the flow of a program based on the value of an int, char or (as of Java 7) an enum value

```
switch (condition) {  
    case condition1:  
        // code  
        break;  
    case condition2:  
        // code  
        break;  
}
```

- Note the break statements. Without them, the switch will execute all code following the matching condition

# Conditionals: Switch

---

- A default case can be added to a switch expression which will be executed when no case is matched

```
switch (condition) {  
    case condition1:  
        // code  
        break;  
    case condition2:  
        // code  
        break;  
    default:  
        // This code will run if neither  
        // condition1 or condition2 is met  
        break;  
}
```



# Loops: While Loop

---

- **Loops** executes block of code a number of times until condition is met
- **While loop** repeat a code block until condition is a strict boolean false

```
int counter = 1;
while (counter < 10) {
    System.out.println(counter);
    counter++;
}
```

- Above, the loop will print numbers from 1 to 9
- While loops have no internal means of keeping track of the number of loops
- Developer must be careful to ensure that infinite loops don't occur

# Loops: While Loop

---

- The conditional in a while loop does not have to be a counter

```
boolean condition1 = true;
while (condition1){
    // code
    if (condition2) {
        condition1 = false;
    }
}
```

- Above, loop executes code block indefinitely because it is looping on a true boolean
- Once **condition2** is met, will swap the loop boolean **condition1** to false
- Last loop is executed

# Loops: Do While

---

- The **do/while loop** similar to while loop, except it executes its code block at least once before checking the condition

```
boolean condition = false;
do {
    // code
} while (condition);
```

- Above, even though condition is immediately set to false, the loop will execute once

# Loops: For Loop

---

- **For loops** are more complex loop that have terminating and increment conditions built in
- Consist of an *initialization block*, a *condition block*, and an *increment block*

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

- Above, an int i is set to zero, it is incremented by one for each iteration of the loop, and once i is greater than ten, the loop terminates.
- Initialized variable is block-scoped to the for loop, i cannot be accessed outside the loop

# Loops: For Loop

---

- Standard form of for loop is the most common, but there are a few variations possible

```
boolean condition = true;
for (int i = 1; condition; i *= 5){
    if (i % 3 == 0){
        condition = false;
    }
    System.out.println(i);
}
```

- Above, an int i is set to one, it is incremented by multiplying it by five each time through the loop, and the loop is broken through some outside condition
- Some specific use cases for unusual for loops like this, but most standard cases require a loop initialized to 1 that increments by one, and ends when a number is reached

# Loops: Nested Loops

---

- Nesting loops can be used to generate two dimensional arrays or tables

```
for (int length = 1; length < 4; length++){  
    for (int width = 1; width < 4; width++){  
        area = length * width;  
        System.out.println("area: " + area);  
    }  
    System.out.println("");  
}
```

- Code above will print a grid that labels the area of rectangles of the given length and width
- Nesting loops is significantly more memory and processor intensive than a single for loop, so be careful with implementations that require them

# Loops: Break and Continue

---

- **Loop-and-a-half** conditions implemented for more precise control of code execution within a loop
- **Break**
  - ◆ Will immediately end all repetitions of a loop and return to normal flow of a program
- **Continue**
  - ◆ Will end the current iteration of a loop, and move on to the next iteration
  - ◆ In *for loop*, will still trigger the increment block
- **Return**
  - ◆ Within a method, return statement can be used to end a loop and return a value; similar to a break statement.
  - ◆ Will end any resources associated with a method, including any further iterations of loop

# WHITE BOARD EXERCISE





```

public class FizzBuzz {

    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("2");
        System.out.println("Fizz");
        System.out.println("4");
        System.out.println("Buzz");
        System.out.println("Fizz");
        System.out.println("7");
        System.out.println("8");
        System.out.println("Fizz");
        System.out.println("Buzz");
        System.out.println("11");
        System.out.println("Fizz");
        System.out.println("13");
        System.out.println("14");
        System.out.println("FizzBuzz");
        System.out.println("16");
        System.out.println("17");
        System.out.println("Fizz");
        System.out.println("19");
        System.out.println("Buzz");
        System.out.println("Fizz");
        System.out.println("22");
        System.out.println("23");
        System.out.println("Fizz");
        System.out.println("Buzz");
        System.out.println("26");
        System.out.println("Fizz");
        System.out.println("28");
        System.out.println("29");
        System.out.println("FizzBuzz");
        System.out.println("31");
        System.out.println("32");
        System.out.println("Fizz");
    }
}

```

Create a method that follows the following rules that :

The method should print out a list of length  $n$ , with each index  $i$  following these rules

If a number  $n$  is divisible by 3, print “Fizz”

If a number  $n$  is divisible by 5, print “Buzz”

If a number  $n$  is divisible by 3 and 5, print “Fizzbuzz”

Bonus :

If a number is prime, do not print it.