



Core Java

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

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

Outline



Prerequisites

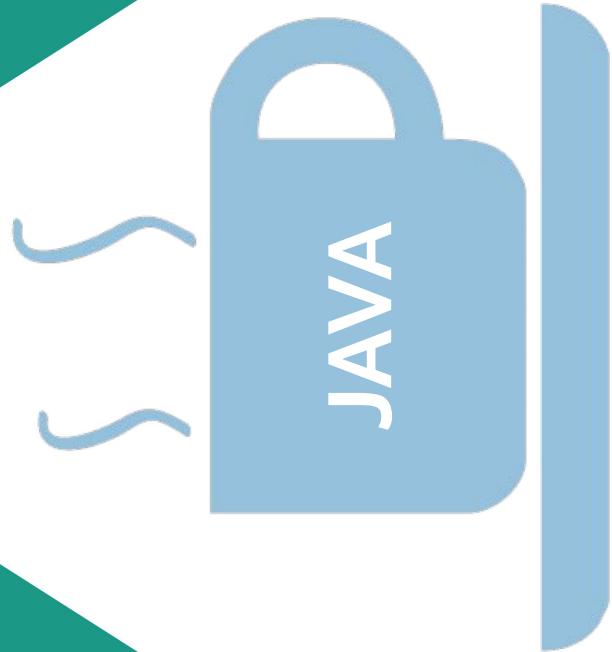
1. JDK Version 8

- 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_Worksplace**” 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 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**

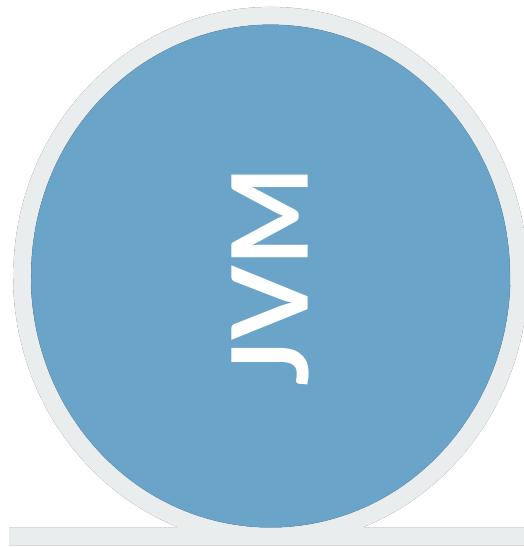


Why Java?

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

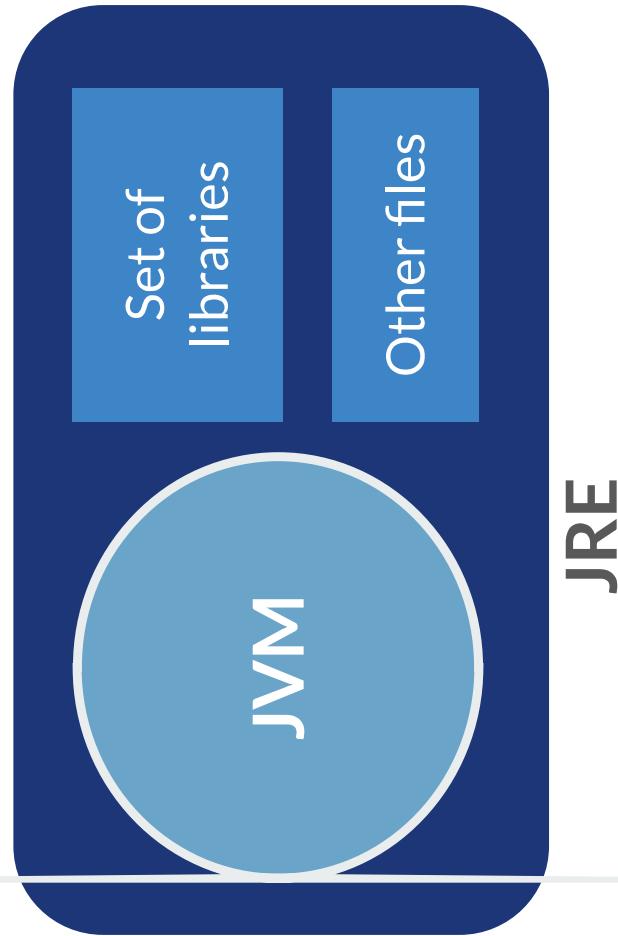
- **Simple syntax**, based in C++ (easier to learn it after C++)
- Removed confusing and/or rarely-used features (explicit pointer, operator overloading etc.)
- No need to remove unreferenced objects, there is **Automatic Garbage Collection**
- **Object-oriented**
- **Architecture neutral**





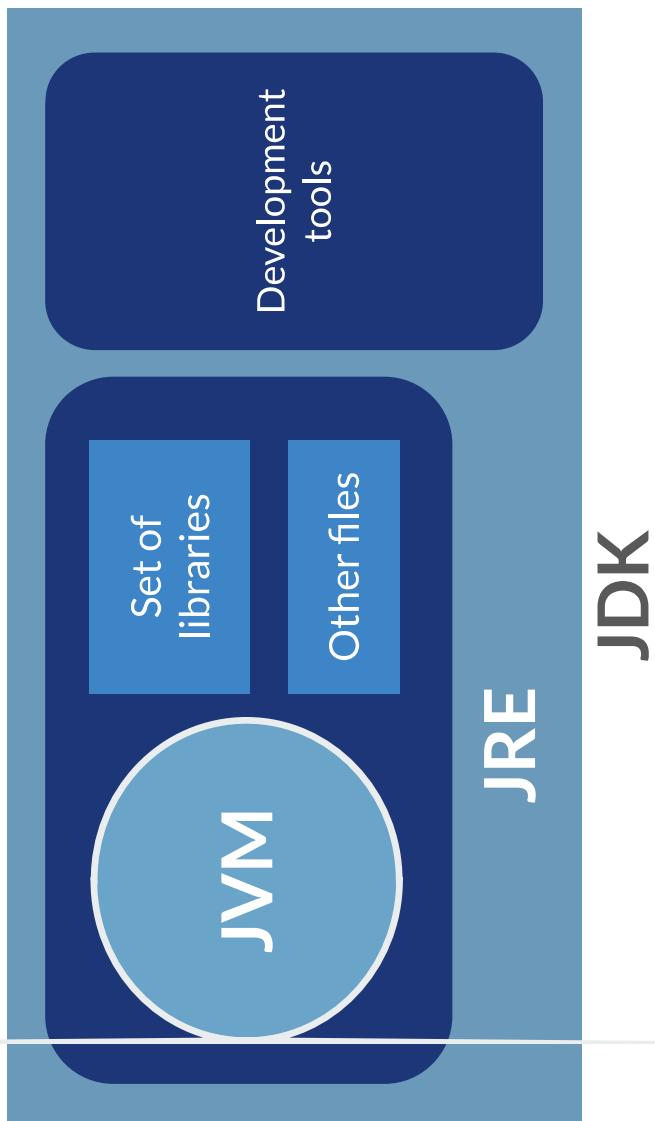
Java Virtual Machine (JVM)

- 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

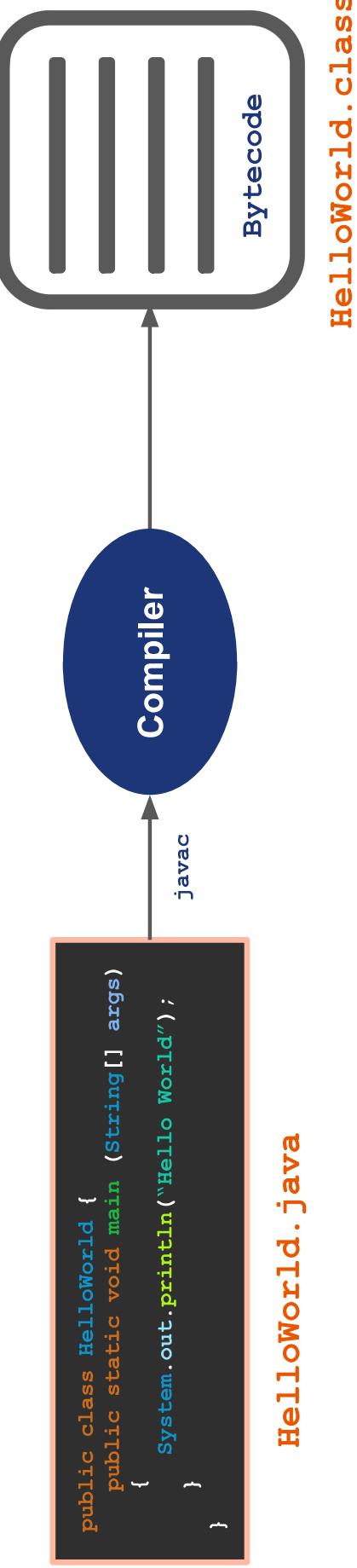
```
package com.cognixia.jump.corejava;

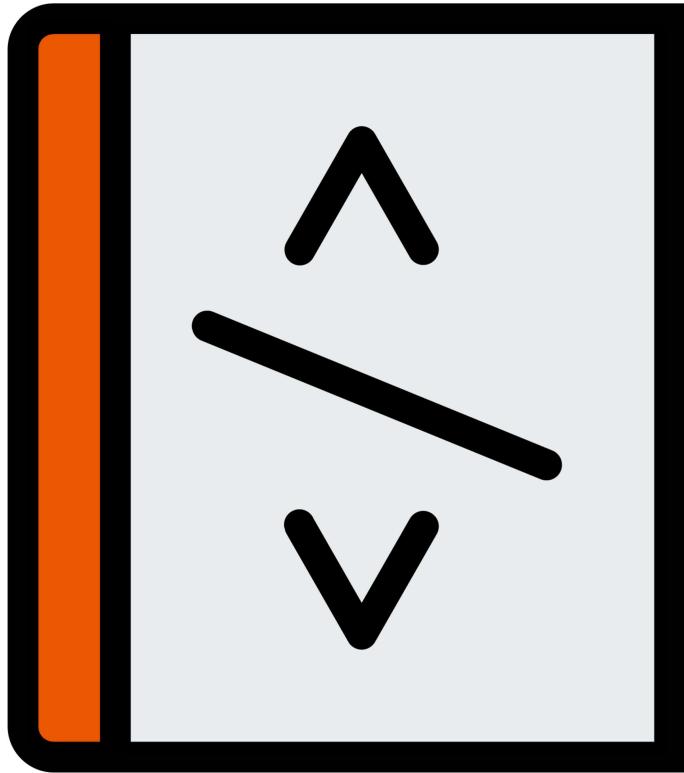
public class HelloWorld {

    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

```
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();  
        ...  
    }  
}
```

Final Keyword

Final - a constant in Java

Final can apply to:

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

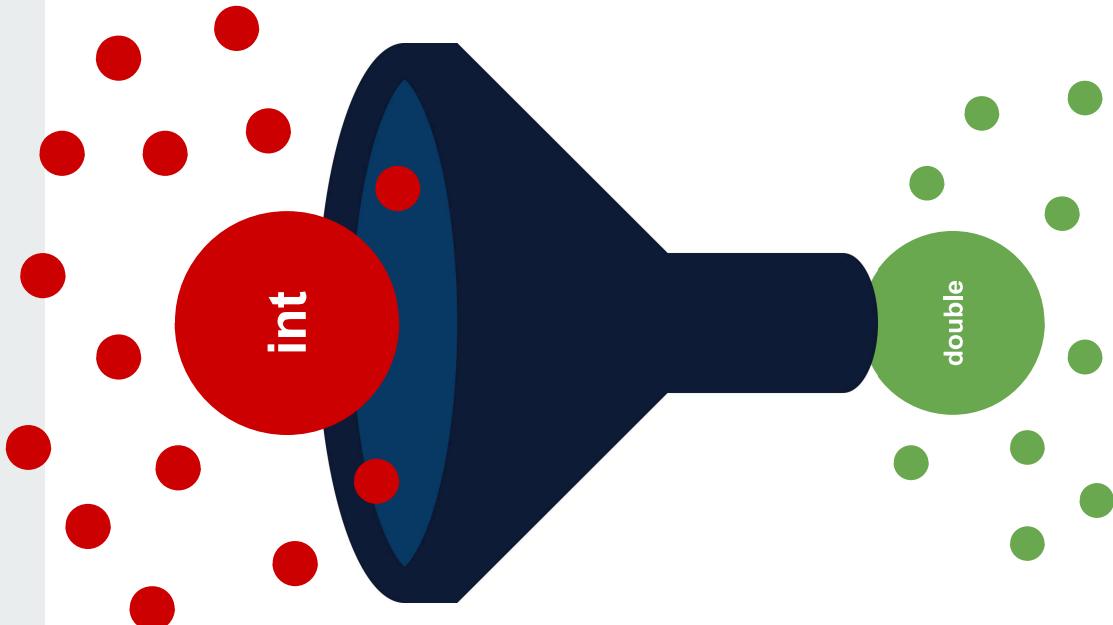


Casting Variables

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:

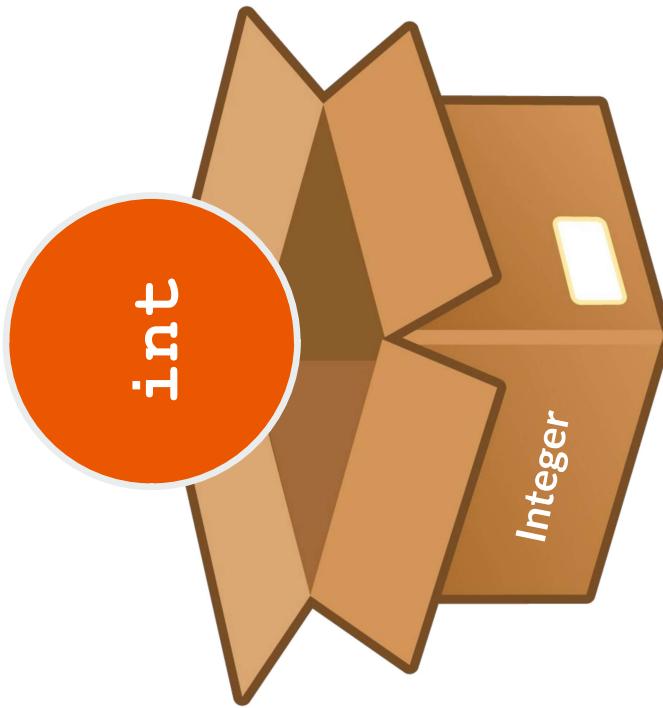
<https://docs.oracle.com/cd/E19253-01/817-6223/chp-typeopexpr-12/index.html>

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

Primitive v Autoboxed

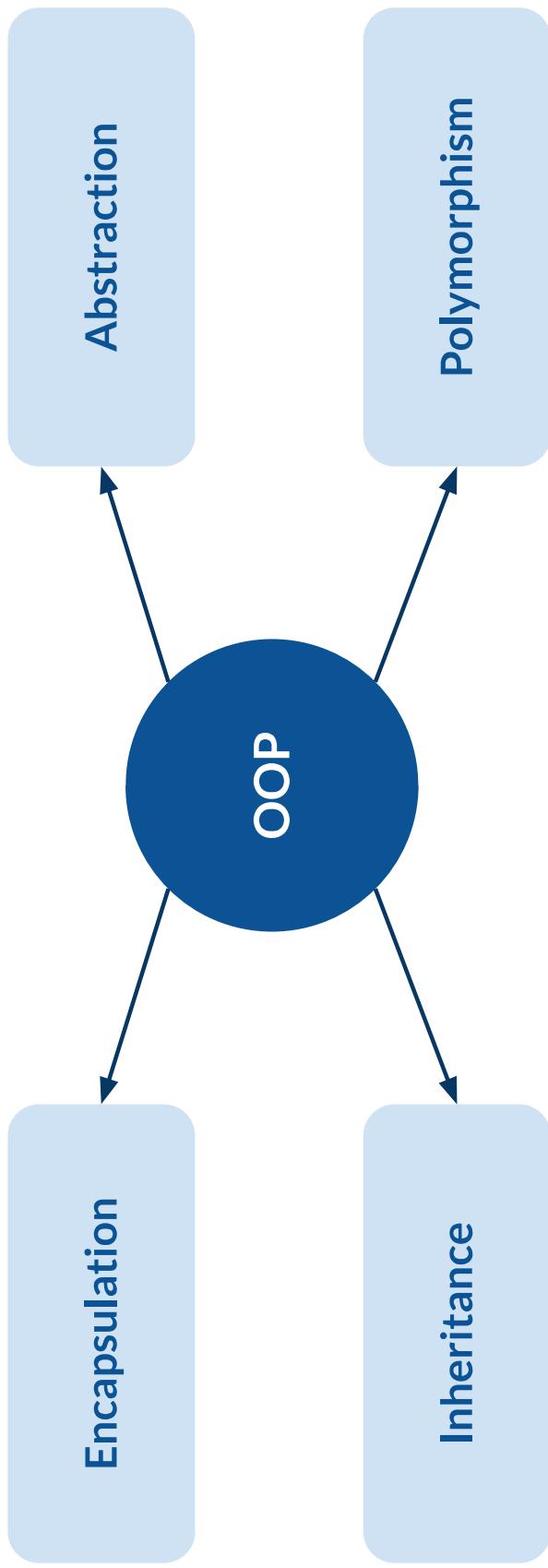
- Java is NOT a pure OOP language
 - ◆ has primitives
- Objects are required in many applications for Java
- **Wrapper Classes** - objects that wrap around the primitive type
- Can use Collections - Collections cannot store primitives on their own

```
double dubs = 5.0;  
int num = (int) dubs;  
  
// passes int to boxed type  
Integer boxed = num;
```



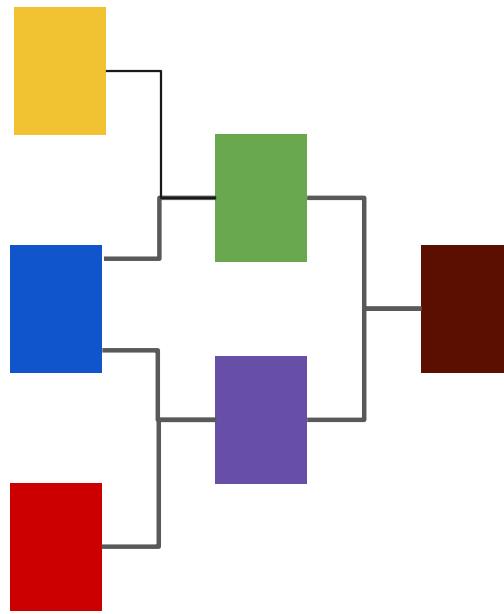


Object Oriented Programming (OOP)

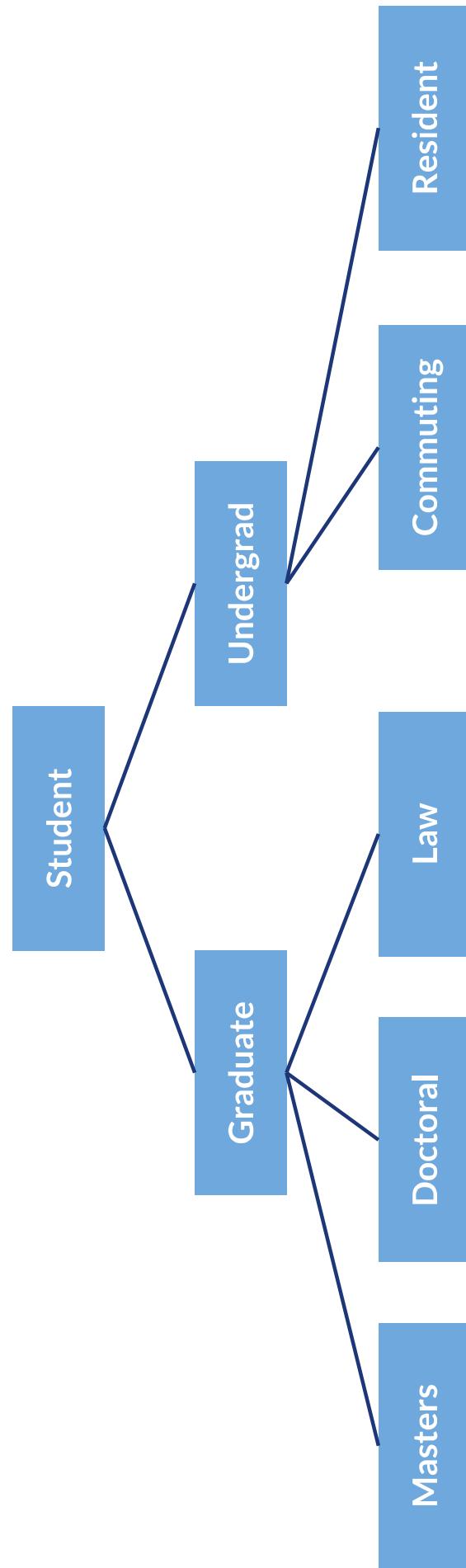


OOP - Inheritance

- **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**

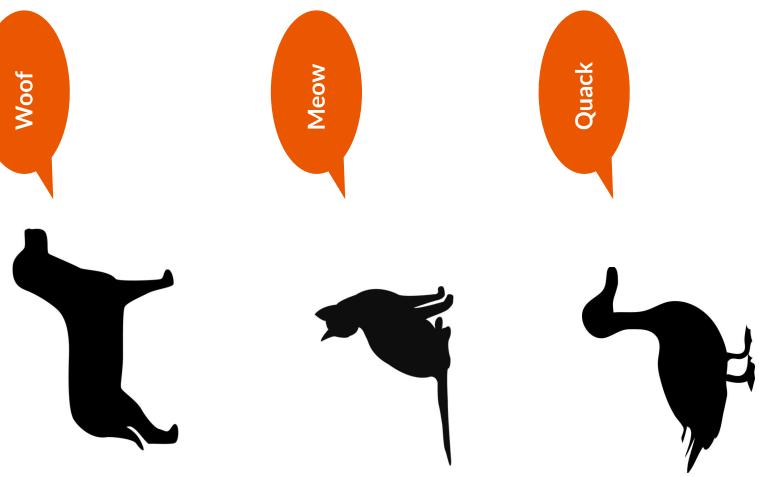


OOP - Inheritance



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 Properties

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.

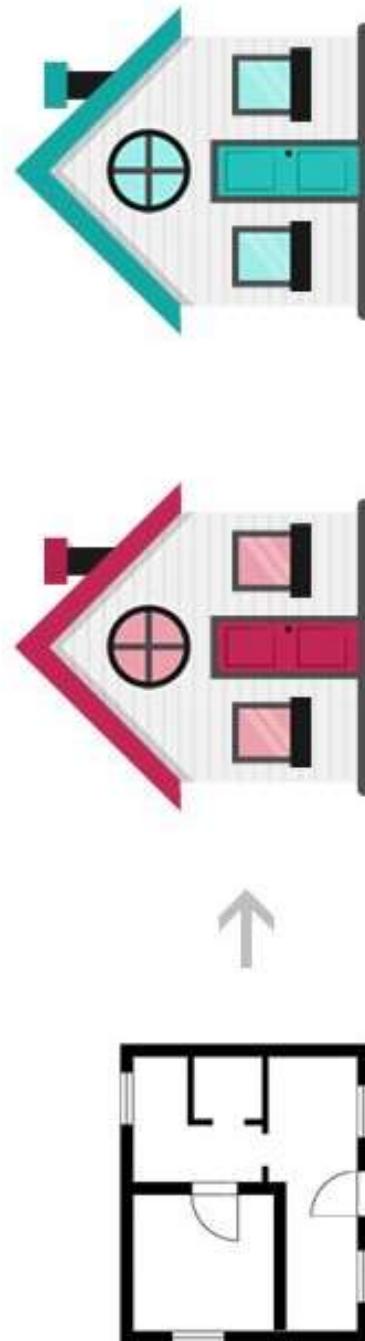
What is happening in
this diagram? Is there
encapsulation?

Child Class Polymorphism

Create a method that
will override one of the
methods from the
parent.

Explain

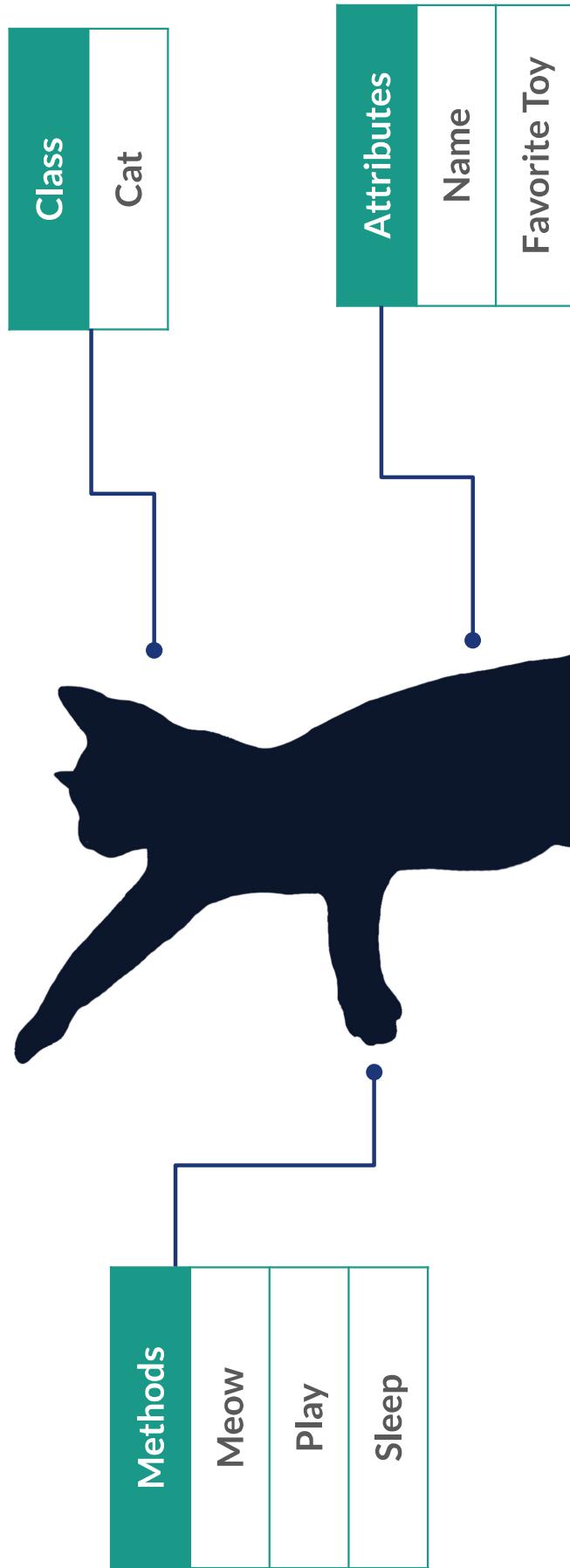
Classes & Objects



Houses built according to the blueprint

Blueprint

Classes



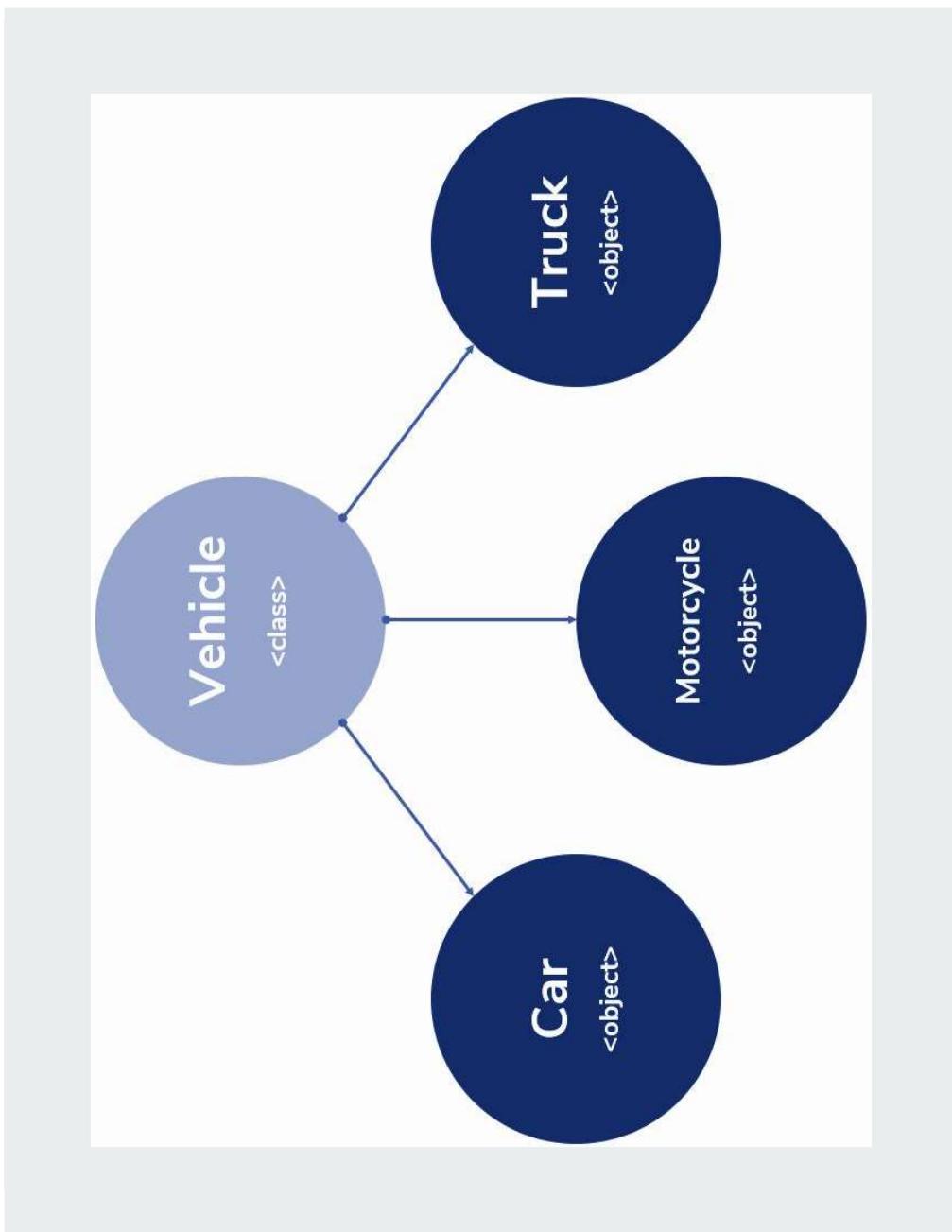
Classes and Objects

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.



```
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

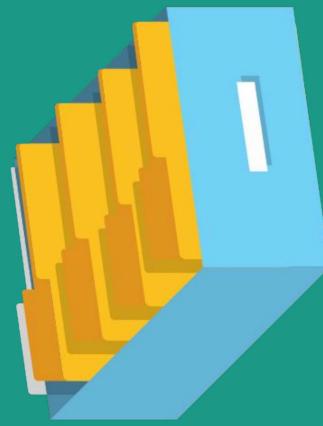
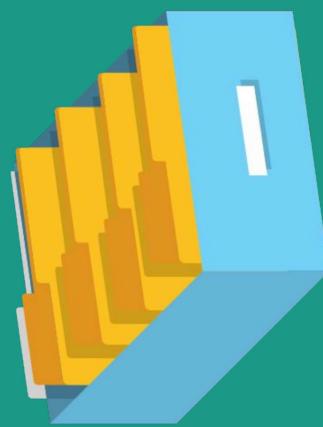
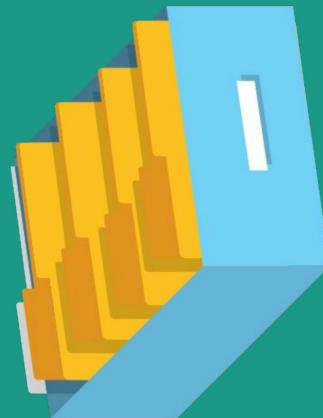
Modifier	Class	Package	Subclass	Global
Public	Allowed	Allowed	Allowed	Allowed
Protected	Allowed	Allowed	Allowed	Denied
Default	Allowed	Allowed	Denied	Denied
Private	Allowed	Denied	Denied	Denied

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:

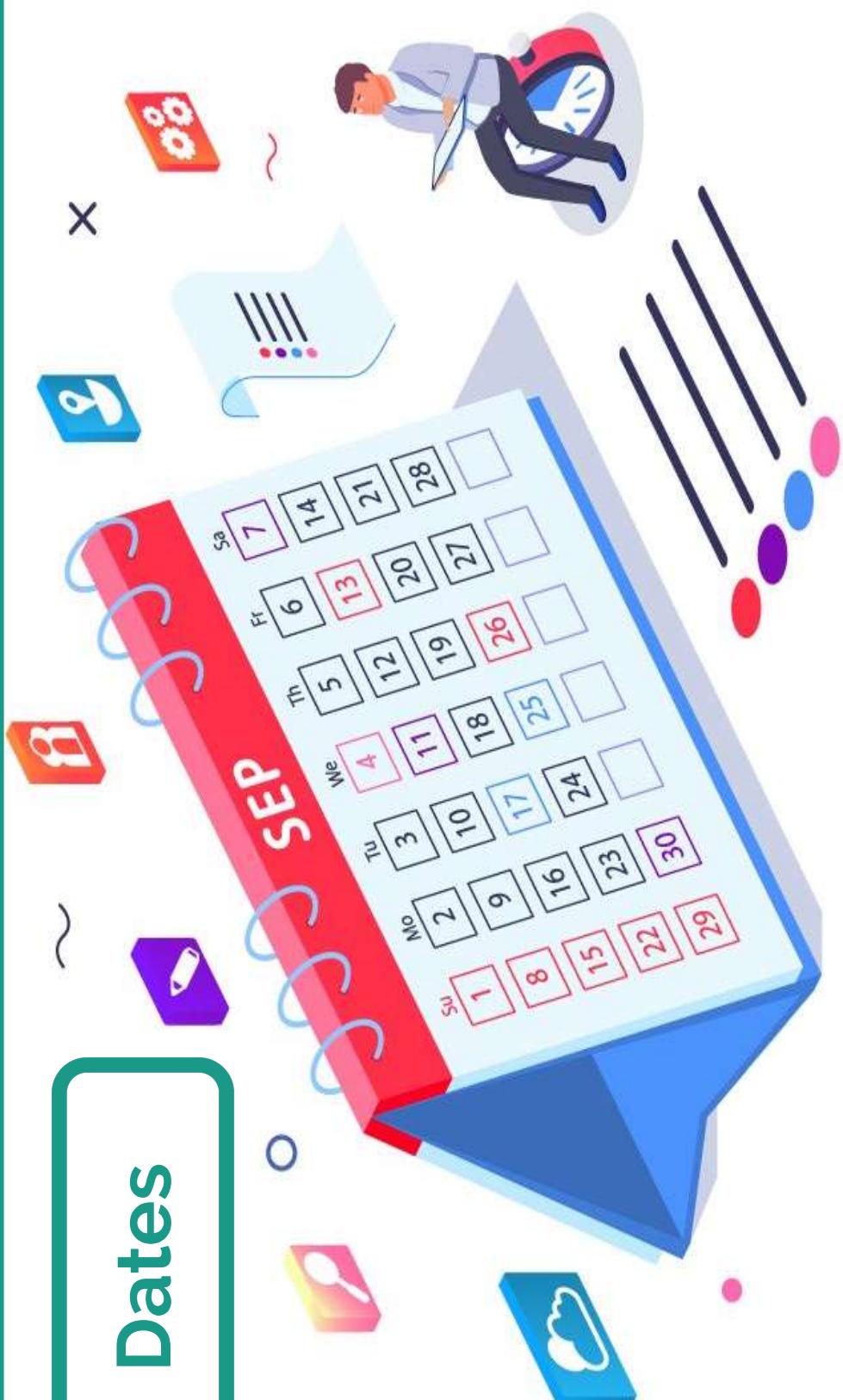
- i.e.- com.cognixia.jump.corejava
- OrganizationType.CompanyName.OrganizationTopic

→ **Imports** - Java needs to know what libraries to reference

to use certain Classes

◆ I.e. - java.lang, java.util

Dates



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  
...
```

LocalDate

```
import java.util.Date;  
import java.time.LocalDateTime;  
  
...  
...  
LocalDate ld;  
ld = LocalDate.of(2015, 7, 3);  
// 2015-07-03  
...  
...
```

- **LocalDate** used to represent a date when time zones do not need to be considered
- By passing in **year, month, and day**; a new LocalDate object is created
- LocalDate gives access to useful methods associated with dates
 - ◆ Adding days, months, or years
 - ◆ Checking the day of the week
 - ◆ Checking if one date is before another

LocalTime

```
import java.util.Date;  
import java.time.LocalDateTime;  
  
...  
LocalTime lt;  
lt = LocalTime.of(8, 45);  
// 2015-07-03T08:45  
...  
...
```

- **LocalTime** used to represent a time without a date
- By passing in an **hour and minute**, a new LocalTime object is created
- LocalTime gives access to useful methods associated with times
 - ◆ Adding hours or minutes
 - ◆ Checking if a time is before another

LocalDateTime

```
import java.util.Date;  
import java.time.LocalDateTime;  
  
...  
LocalDateTime ldt;  
ldt = LocalDateTime.of(2015, 7, 3, 8, 45);  
// 2015-07-03T08:45  
...  
...
```

- **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

```
import java.util.Date;
import java.time.LocalDateTime;
import java.time.ZonedDateTime;
...
LocalDateTime ldt;
ZonedDateTime zdt;
ZonedDateTime 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]
...

```

- **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.

Convert the following Strings to Date objects:

Computer science student



04/23/2004

November 05 2014

1993/30/09

Assume the following Dates and Times on the left are in the EST time zone. Convert them to the given time zones on the right.

Senior developer, 10+ years experience



04/23/2004 7:30pm



November 05 2014 9:23am



Gurugam

1993/30/09 21:50



Honolulu





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 compared memory locations)
- != → Not equal to
- ! → Not (reverses a boolean)
- && → And (True if Both booleans are true)
- || → Not (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 with 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 10
- 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



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”**

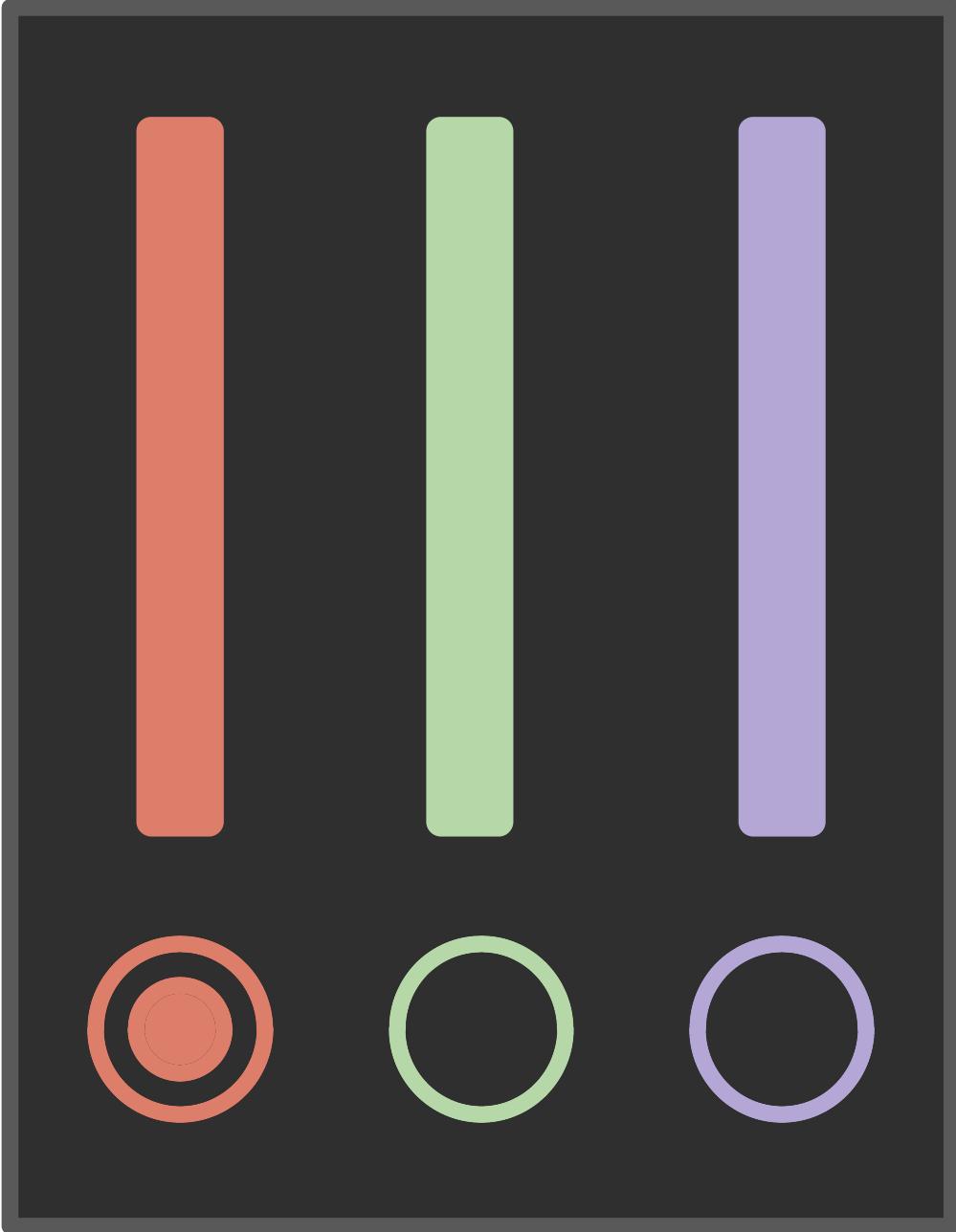
**If a number n is divisible by 5, print “Buzz”**

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

Bonus :

**If a number n is prime, do not print it.**

```
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("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("35");
}
```



Enums

# Enums

- **Enumerated Constants or Enums**
  - are a Java language supported way to create constant values
  - More type safe than variables like String or int
  - If used properly, enums help create reliable and robust programs

```
public enum Days {
 MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
 FRIDAY, SATURDAY, SUNDAYDAY
}

public enum Months {
 JANUARY, FEBRUARY, MARCH, APRIL, MAY,
 JUNE, JULY, AUGUST, SEPTEMBER, CYBER,
 NOVEMBER, DECEMBER
}
```

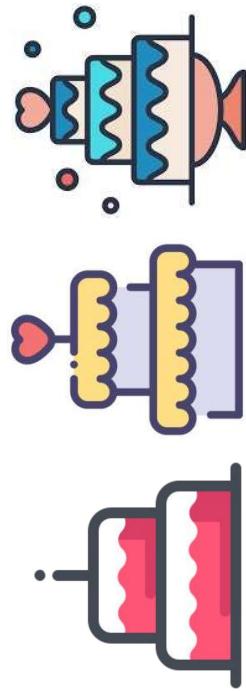
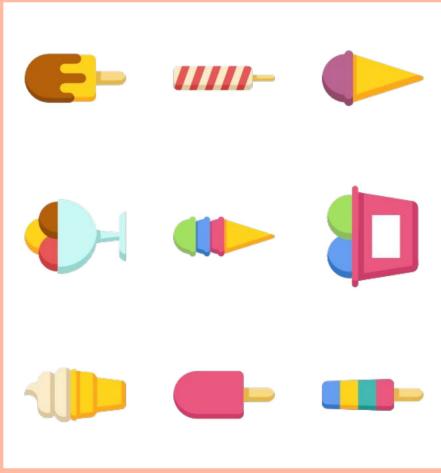
# Enums

- Enums create public static final int variables.
- Without enums:

```
public static final int ICE_CREAM = 0;
public static final int CHOCOLATE = 1;
public static final int VANILLA = 2;
```

- With enums:

```
public static enum Cake {
 ICE_CREAM, CHOCOLATE, VANILLA
}
```



# Enums

---

- Once an enum is created, its values can be accessed with dot notation
- A local variable can be instantiated of the enum's type, then initialized with a value from the enum :

```
public static enum Cake {
 ICE_CREAM, CHOCOLATE, VANILLA
}
...
Cake c1 = Cake.CHOCOLATE;
System.out.println(c1); // CHOCOLATE
```

# Enums in Switches

- Enums can be used in switch cases.
- Enums allow for switch cases to use values no more complex than int or char

```
Cake favCake = Cake.CHOCOLATE;
switch (favCake) {
 case ICE_CREAM:
 ...
 break;
 case CHOCOLATE:
 ...
 break;
 case VANILLA:
 ...
 break;
}
```

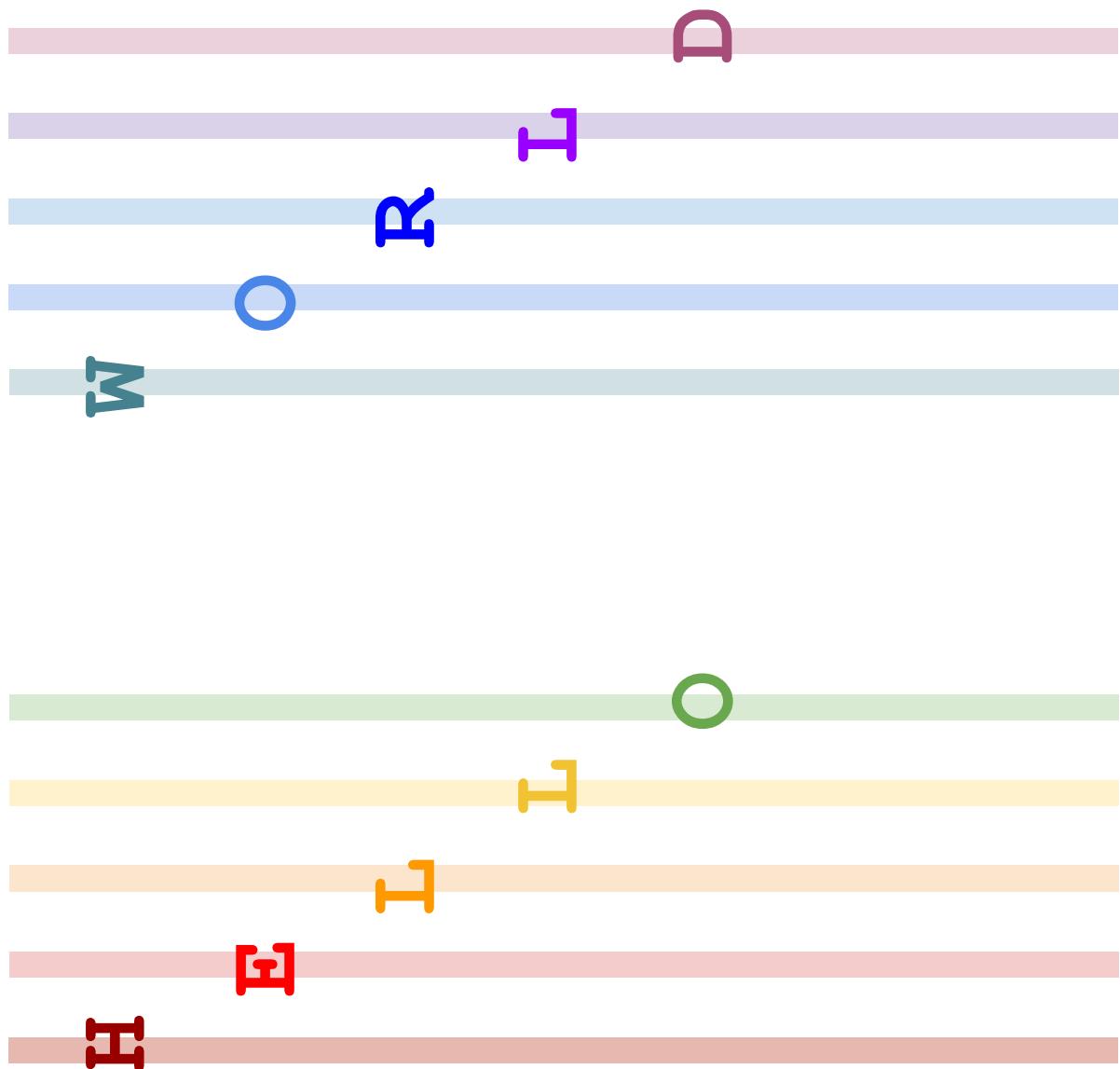
## Enums with Constructors

- Each value in an enum instantiates a public final class
- The constructor defined in the enum is run for each value
- Arguments passed to the enumerated values can be used in the constructor to create new data members

```
public enum Cake {
 ICE_CREAM("Graham Cracker"),
 CHOCOLATE("Chocolate Custard"),
 VANILLA ("Fresh Strawberries");

 public final String filling;

 Cake (String filling) {
 this.filling = filling;
 }
}
```



Strings

# Strings

- **Strings** are not primitive data types, they are objects
- When creating a string, a new object is created with the name assigned to its memory location
- Strings are so ubiquitous that, they are often considered alongside primitives even though they are objects

```
String str = "Hello World";
```

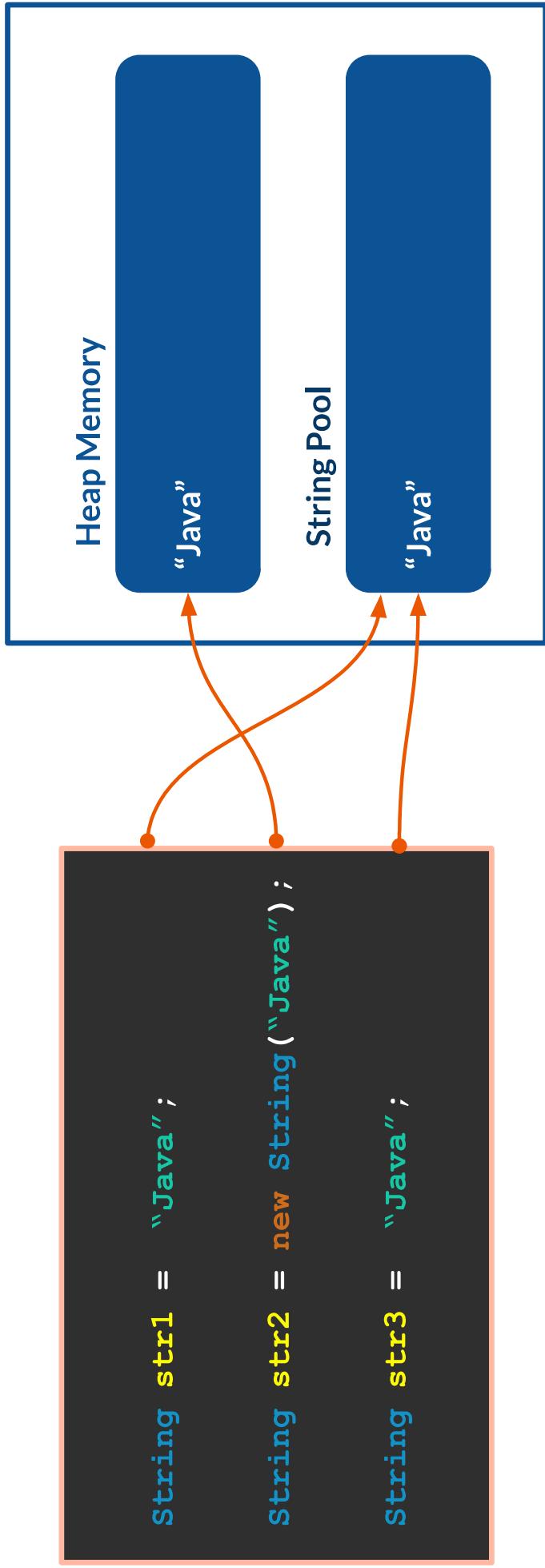


# String Immutability and String Pool

---

- Strings are *final and immutable*
  - ◆ Once created, a string object cannot be altered; such as adding or removing a character.
- String's value is reassigned, whole new string object created
- **String Pool** is a place in memory where the JVM stores each unique character sequence that is created
- When a new string is created, JVM first searches for that char sequence in the String Pool
  - ◆ If it finds the sequence, assigns the new object that value in the pool
  - ◆ If not, first creates a string literal of the char sequence in String Pool, then assigns the variable to that memory location
- Using the new `String("Hello World")` syntax will always create two new strings, one in the heap memory, and one in the pool

# String Pool Diagram



# Comparing Strings

---

- Consider the following scenario:

```
String str1 = "Java";
String str2 = new String("Java");
```
- Two strings have the same char sequence. However, if we try and compare them:

```
str1 == str1; // true
str2 == str1; // false
```
- Strings are objects, so the variables we assign them are references to memory locations, not the char sequences themselves.
- To check the equivalence of strings, we must use the .equals() method:

```
str1.equals(str1); // true
str2.equals(str1); // true
```

# String Methods

---

There are almost 50 methods defined in the `String` class. Important methods include:

- **Length:**
  - ◆ Returns the number of chars in the string as an int
- **CharAt:**
  - ◆ Returns the char at the given index
- **IndexOf:**
  - ◆ Returns the first index of a given char in the string
- **Substring:**
  - ◆ Extracts a section of the string at the given indexes
- String **Concatenation**, while not a defined method, is an important operation when working with strings

# Length of a String

---

| Character: | H | e | I | l | o |   | W | o | r | l | d  |
|------------|---|---|---|---|---|---|---|---|---|---|----|
| Index:     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

- Strings are arrays of characters (more on arrays later)
  - The `.length()` method will return the total number of characters in the array
- ```
System.out.println("string length: " + str.length());  
// string length: 11
```
- Important: Array indexes start at 0, so the length of the string is one more than the last index

Single Character Access

Character:	H	e	I	l	o		W	o	r	1	d
Index:	0	1	2	3	4	5	6	7	8	9	10

- Individual characters can be accessed with the .charAt() method:

```
System.out.println("first letter: " + str.charAt(0));  
// string length: 11
```

- Trying to access a char outside the length of an array results in an error

Finding an Index

Character:	0	1	2	3	4	5	6	7	8	9	10
Index:	H	e	l	l	o		W	o	r	l	d

- The reverse operation of `.charAt()` is also possible
- The `.indexOf()` method will return the index where a given substring appears in the string
- `.indexOf()` is an overloaded operator that can accept a string or a char; and it can take a second argument of a starting index

```
String s = "first letter 'H': " + str.indexOf('H'); // first letter 'H': 0
String s = "first 'W': " + str.indexOf('W'); // first letter 'W': 6
String s = "second letter 'o': " + str.charAt('o', 5); // second letter 'o': 7
```

Substrings

Character:	H	e	l	l	o		W	o	r	l	d
Index:	0	1	2	3	4	5	6	7	8	9	10

- The `.substring()` method allows a developer to get access to only a specific part of string
- `.substring()` returns a new string, original string is not modified
- `.substring()` is an overloaded operator that accepts either one or two integers
 - ◆ `.substring(i,j)` will return a new string starting with the character at i and ending before index j
 - ◆ `.substring(i)` will return a new string starting at index i and containing the rest of the string

String Concatenation

- Strings can be combined with the “+” operator:

```
String str1 = "Hello";
String str2 = " World";
String str = str1 + str2; //Hello World
```

- the data types can be concatenated with strings

```
str = str + 3.14; // Hello World3.14
```

- However, the way that the plus operator is used depends on order

```
str = 3.14 + 7 + str; // 10.14Hello World
```

- “+” is an operator; concatenation is not a method in the string class
- Concatenating does not alter the original strings

Other String Methods

- The String class gives us access to some very useful methods
 - **.compareTo()**
 - ◆ Compares two strings lexicographically
 - **.trim()**
 - ◆ Removes leading and trailing whitespace characters
 - **.valueOf()**
 - ◆ Converts a primitive to a string
 - **.startsWith()**
 - ◆ Returns true if the string starts with the given char
 - **.endsWith()**
 - ◆ Returns true if the string ends with the given char
- Most IDEs will give a user access to all available methods for a given class.
 - ◆ All string methods are detailed in the Java Documentation
 - ◆ <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/String.html>

StringBuffer and StringBuilder

- To work with character sequences which are mutable, the StringBuffer or the newer StringBuilder (Java 5) classes are used.
 - ◆ StringBuffer is thread safe, but Stringbuilder is faster.
 - ◆ Both StringBuffer and StringBuffer have the same methods, so they are interchangeable
- StringBuffer and StringBuilder can:
 - ◆ Replace chars at a given index
 - ◆ Append or prepend any primitive data type to a string
 - ◆ Insert a new char sequence at a given position

```
StringBuffer sb = new StringBuffer("Hello") ;  
sb.append(" World"); // Hello World  
sb.setCharAt(5, '@'); // Hello@World  
sb.append(1); // Hello@World1  
sb.append(true); // Hello@World1true
```

White board Exercise



Coding challenge:

Create a method that takes a string, reverses it, and returns the reversed string. You can use a built in method to do this

Bonus 1:

Create a second method that does not use any built in methods to reverse the string.

Bonus 2:

For a given string with multiple words, reverse each word individually



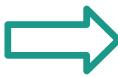
Regular Expressions

The universal language for character pattern matching

REGEX: Pattern Matching

A **regular expression** is a string that describes a search pattern. They can be used to find certain substring patterns in a string or used to validate user input.

`^|(\d{3})\s?\d{3}-\d{4}$`



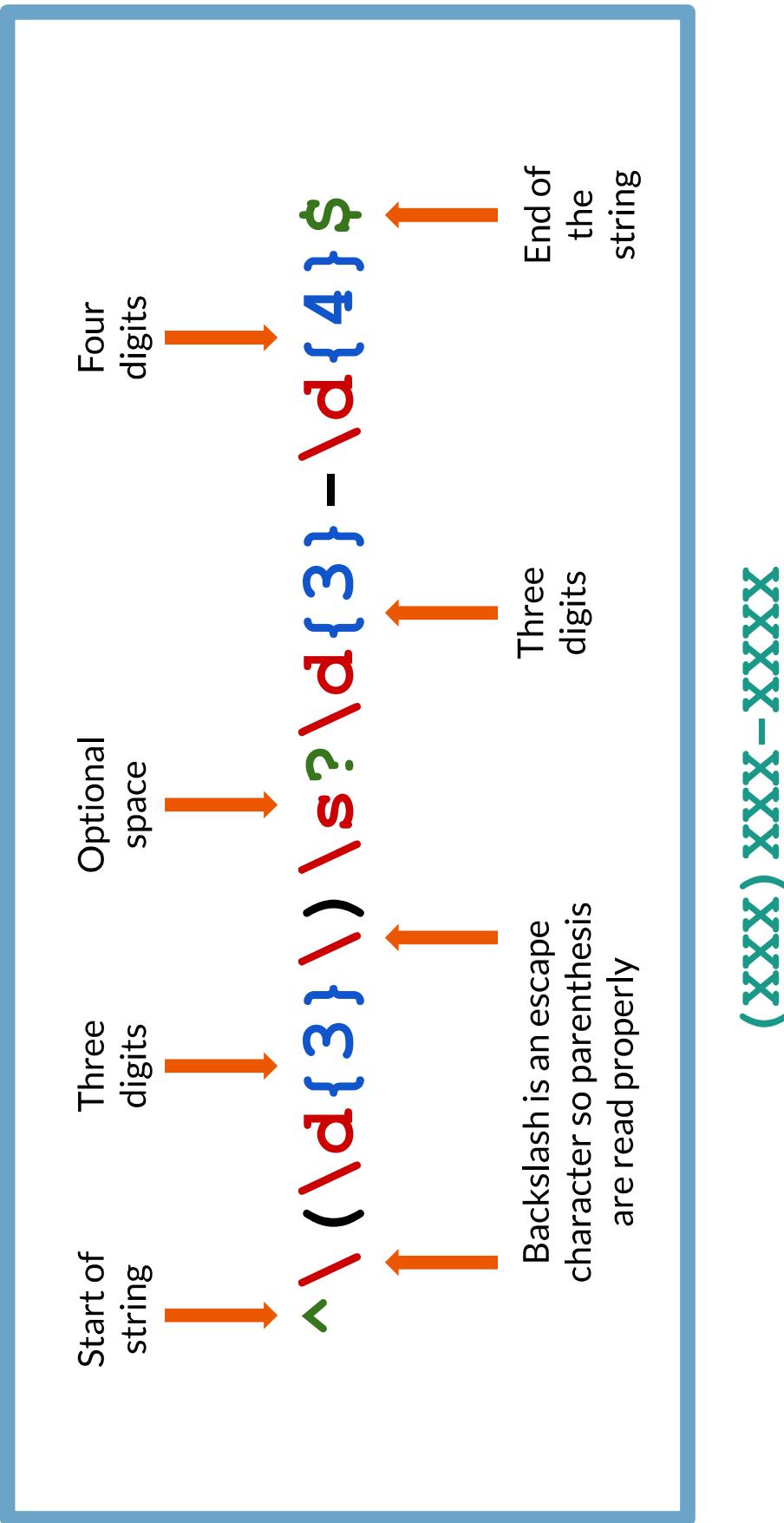
`(012) 345-6789` ✓



^	If first in the regex, denotes the start or as a negation	^Hello → string must start with word “Hello” [^a] → string cannot have the character “a”
\$	Denotes the end of a string	Hello\$ → string must end with word “Hello”
*	Zero or more occurrences	ba*b → string has two b’s with zero or more a’s between them
+	One or more occurrences	b(ac)+b → string has two b’s with one “ac” r multiple “ac” strings between them
?	Zero or one occurrence	bc?b → string has two b’s with nothing between them or a single c
{}	Matches a string in the range specified	abc{2, 5} → string starts with ab and follows with two to five c’s
() and 	Used to provide section of choices	b(a i e)t → string is either bat, bit, or bet
[]	Same as above	b[aie]t → string is either bat, bit, or bet

.	Any character, can be a alphanumeric or a symbol	b.c → string can bac, b5c, b@c, etc.
[0-9]	Digit from 0 t 9	[0-9] [0-9] → string from 00 to 99
[a-z]	Lowercase letter, switch to capital A and Z for all uppercase letters	[a-z] + → string with a lowercase characters repeated one or more times
[a-zA-Z]	A letter, lowercase or uppercase	[a-zA-Z] {2} → a two character string with any letter (wE, BP, ee...)
\w	A word character: letter, number, or an underscore	\w@gmail.com → hello@gmail.com, 123@gmail.com, b_tr@gmail.com
\d	A digit	\d* → string with zero or more digits (4, 48, 489..)
\s	Whitespace character that includes tabs and line breaks	hello\sworld → string has a whitespace character like a tab or linebreak between the two words

Additional resources [here](#) and [here](#).



Reg(Ex) Exercise



Please close your laptops, have a paper and writing utensil out. Make sure you have received the regex reference sheet. When you have your answer, raise your hand.

Write a regex expression for an address. For this exercise we will assume that an address is formatted like this:

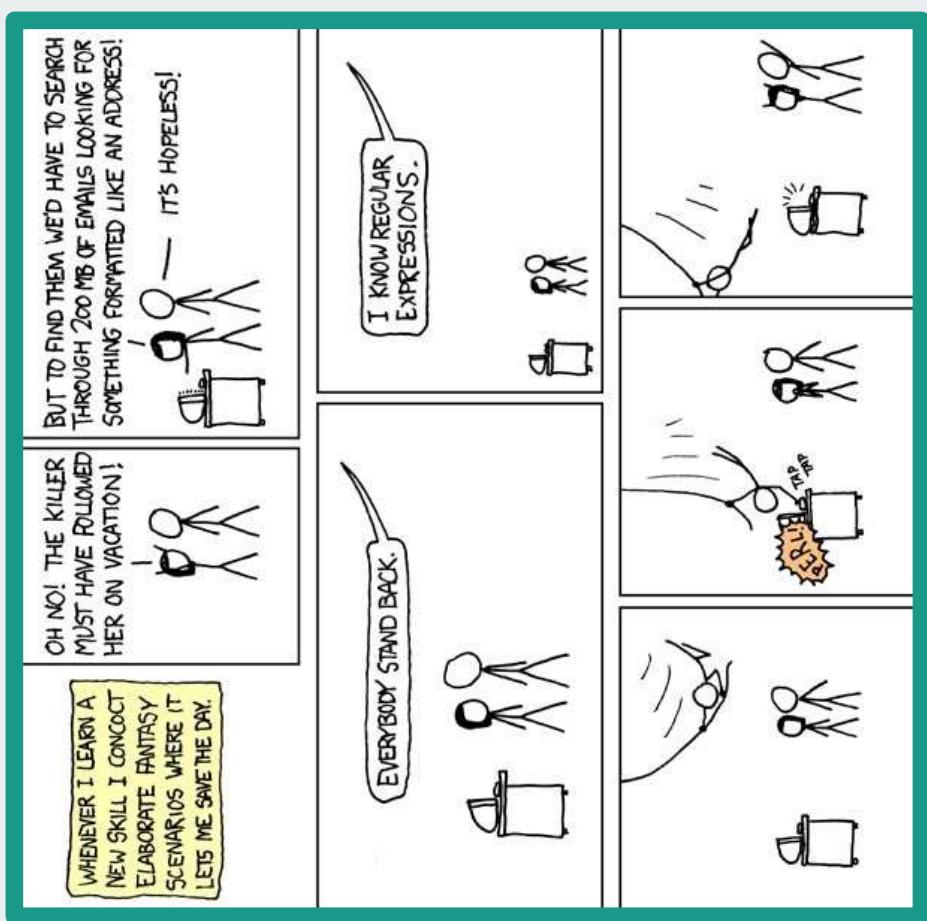
1 Some Street, City, ST 12345

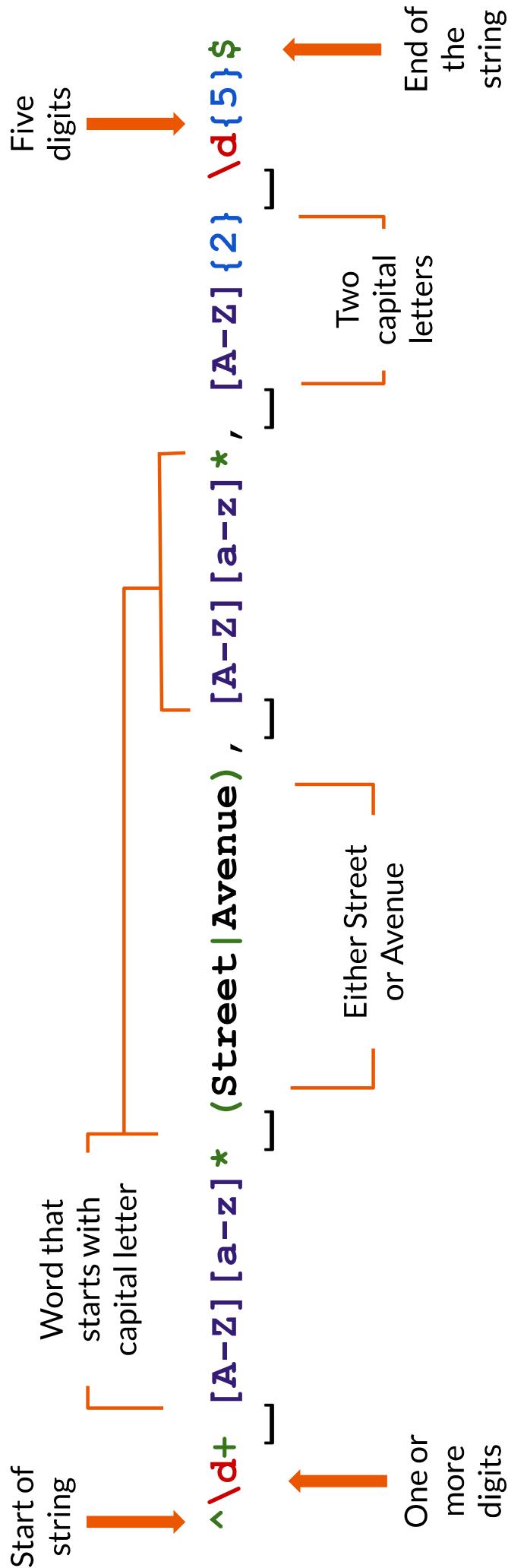
The street just has to lead with a number.

Street name should be two words with only letters. The second word of the street name should be either "Street" or "Avenue". The city should be one word. The state should be two capital letters, doesn't have to be a real state. The zip can just be exactly 5 digits long. Have a comma before and after the city.

25 Water Avenue, Bee, BZ 45092

9 South 27th Place, A City, CAB 12378-0123





Pattern & Matcher

- **Pattern** = compile representation of a regular expression
- **Matcher** = created from a Pattern, matches a regular expression against a string
- Use **matches()** vs **find()** for Matcher depending on if you want to match regex to entire string or part of it

```
String regex = "ba*b";  
Pattern pattern = Pattern.compile(regex);  
  
String str = "baab";  
  
Matcher matcher = pattern.matcher(str);  
  
System.out.println("String matches: " +  
    matcher.matches());
```