

Java Fundamentals

Angie Jones

Certified Java Programmer and Java Champion

Set Up

- IntelliJ Community Edition
  https://www.jetbrains.com/idea/download/

- Java 17 or later
  IntelliJ > Download JDK

- GitHub repo
  https://github.com/angiejones/frontendmasters-java-fundamentals



The Basics

let's write our first program to learn the basics



Gross Pay Calculator

Write a program that calculates an employee's gross pay



Decision Structures

programs often have to evaluate conditions to determine which blocks of code to execute

If Statements

If a certain situation occurs,
do <something>,
then go back to the main flow

Salary Calculator

All salespeople get a payment of $1000 for the week.

Salespeople who exceed 10 sales get an additional bonus of $250.

If-Else Statements

If a certain situation occurs, <do something>.

Otherwise <do something else>.



Quota Calculator

All salespeople are expected to make at least 10 sales each week. For those who do, they receive a congratulatory message. For those who don't, they are informed of how many sales they were short.

Nested If Statements



Path inside of a path

If a certain situation occurs, check for the next situation

Nested If Statements

```
1 if(salary >= requiredSalary) {
2
3     if(years >= requiredYearsEmployed){
4         System.out.println("You qualify for the loan.");
5     }
6     else{
7        System.out.println("Sorry, you do not qualify.");
8     }
9 }
```

If-Else-If Statement

If situation A occurs, <do something>

Else if situation B occurs, <do something else>

Else if situation C occurs, <do something else>



Test Results

Display a letter grade for a student based on their test score.

Switch Statements

- solves problem in the same way that if-else-if does

- good for when there's more than 2 possible paths

- each path checks for equality





Grade Message

Have a user enter their letter grade, and using a switch statement, print out a message letting them know how they did.

Switch Expressions

Similar to switch statements but allow you to directly assign a value when a case is matched.



Grade Message

Have a user enter their letter grade, and using a switch expression, print out a message letting them know how they did.

Relational Operators

OPERATOR	MEANING	EXAMPLE
>	greater than	2 > 3 -> false
<	less than	2 < 3 -> true
>=	greater than or equal to	4 >= 4 -> true
<=	less than or equal to	4 <= 3 -> false
==	equal to	3 == 2 -> false
!=	not equal to	3 != 2 -> true



Equality Operator (==)

single equal sign is for assignment

double equal sign is to check for equality

Logical Operators

SYMBOL	OPERATOR	TO BE TRUE	EXAMPLE
&&	AND	both conditions must be true	1 <= 2 && 4 != 5
||	OR	at least one condition must be true	3 == 4 || 2 == 2
!	NOT	condition must be false	!(2 == 3)

Nested If Statements

```
1 if(salary >= requiredSalary) {
2
3     if(years >= requiredYearsEmployed){
4         System.out.println("You qualify for the loan.");
5     }
6     else{
7        System.out.println("Sorry, you do not qualify.");
8     }
9 }
```

Logical Operators

```java
if(salary >= requiredSalary && years >= requiredYears){
    System.out.println("You qualify for the loan.");
}
else{
    System.out.println("Sorry, you do not qualify.");
}
```



Repetition Structures

loops are structures that cause a block of code to repeat



Input Validation

Each store employee makes $15 an hour. Write a program that allows the employee to enter the number of hours worked for the week. Do not allow overtime.

While Loops

Condition Controlled

continues running while the specified condition remains true

Pre-test

condition is tested before entering the loop

Execution

use when loop may or may not need to be executed



Add Numbers

Write a program that allows a user to enter two numbers, and then sums up the two numbers. The user should be able to repeat this action until they indicate they are done.

Do While Loops

Condition Controlled

continues running while the specified condition remains true

Post-test

condition is tested after completing the loop

Execution

use when loop should run at least once, and possibly more



Cashier

Write a cashier program that will scan a given number of items and tally the cost.

For Loops

Count Controlled

runs a specified number of times

Pre-test

condition is tested before entering the loop

Execution

use when you know how many times the loop should be executed

Nested Loops

sometimes your repetitive tasks also contain repetitive subtasks

Nested Loops

Find the average test scores for each student in the class.

Loop 1: go through every student in the class

Loop 2: go through every test grade that student has

```java
/*
 * NESTED LOOPS:
 * Find the average of each student's test scores
 */
public class AverageTestScores {

    public static void main(String args[]){

        //Initialize what we know
        int numberOfStudents = 24;
        int numberOfTests = 4;

        Scanner scanner = new Scanner(System.in);

        //Process all students
        for(int i=0; i< numberOfStudents; i++){

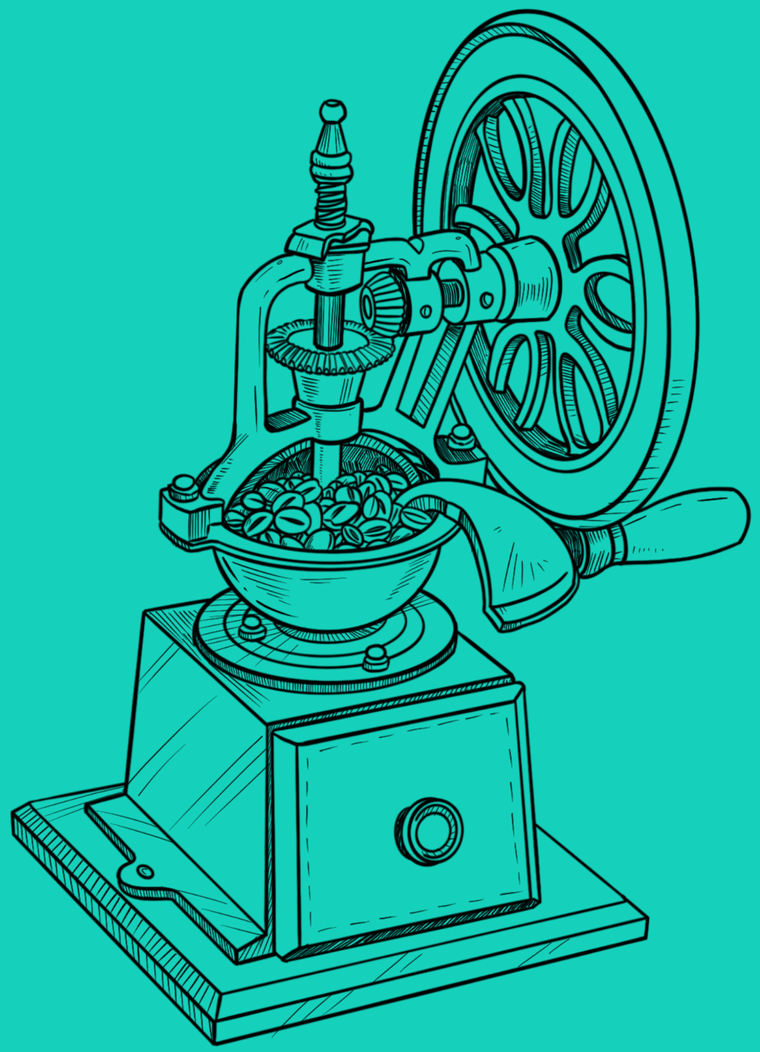

        }

        scanner.close();
    }
}
```

```
15 //Process all students
16 for(int i=0; i< numberOfStudents; i++){
17
18     double total = 0;
19
20     //Process student's tests
21     for(int j=0; j<numberOfTests; j++){
22
23     }
24
25 }
```

```
18 double total = 0;
19
20 //Process student's tests
21 for(int j=0; j<numberOfTests; j++){
22     System.out.println("Score for Test #" + (j+1));
23     double score = scanner.nextDouble();
24     total = total + score;
25 }
```

```
15 //Process all students
16 for(int i=0; i< numberOfStudents; i++){
17
18     double total = 0;
19
20     //Process student's tests
21     for(int j=0; j<numberOfTests; j++){
22         System.out.println("Score for Test #" + (j+1));
23         double score = scanner.nextDouble();
24         total = total + score;
25     }
26
27     double average = total/numberOfTests;
28     System.out.println("The test average for student #" +
   (i+1) + " is " + average);
29 }
```



Methods

collection of statements that perform a task

Methods

```
1 public static int calculateSum(int number1, int number2) {
2
3 }
```

Methods

```
1 public static int calculateSum(int number1, int number2) {
2
3   int sum = number1 + number2;
4
5 }
```

Methods

```java
public static int calculateSum(int number1, int number2) {

   int sum = number1 + number2;
   return sum;
}
```



Greet User

Write a method that asks a user for their name, then another method that greets the user by name.

Overloaded Methods

A class can have multiple methods with the same name, but they must have unique parameter lists

Overloaded Methods

```java
public class Month {

  public int getMonth(String month) {
    //...
  }

  public String getMonth(int month) {
    //...
  }
}
```

Overloaded Methods

```java
public class Month {

  public String getMonth(int month) {
    //...
  }

  public int getMonth(String month) {
    //...
  }


  public int getMonth(String monthName) {
    //...
  }
}
```



Variable Scope

A variable is only accessible within the scope it is declared within



Instant Credit Check

Write a program that approves anyone who makes more than $25,000 and has a credit score of 700 or better. Reject all others.

Local Variable Type Inference

when declaring a local variable as var,

Java will infer its type

Local Variable Type Inference

```
boolean isWaterWet = true;
```

Local Variable Type Inference

```
var isWaterWet = true;
```

Local Variable Type Inference

```
public static void main(String[] args) {
    var isWaterWet = true;
}
```

Local Variable Type Inference

```
public class MyClass {

  var isWaterWet = true;

  public static void main(String[] args) {}

}
```



Objects

objects are structures which contain data and behaviors.



Rectangle

Write a class that describes the fields and methods of a rectangle.

Encapsulation

All data and behavior is contained within the object itself.

Expose behavior and restrict direct access to data.

Constructors

Structures within a class that are used to set the initial state of an object.



Constructors

Add a default constructor and an all-arg constructor to the Rectangle class

Instantiating Objects

Create an instance of the Rectangle class to represent an object of that type



Area of Rooms

Write a class that creates instances of Rectangle to find the total area of two rooms in a house.

Sending & Receiving Objects

Objects can be passed into methods and returned from methods



Area of Rooms

Redo HomeAreaCalculator to include a calculateArea() method that accepts two Rectangles and a getRoom() method that returns a Rectangle.

Records

A special type of Java class that's great for simple objects

Creating a Record

```
1 public record Account(
2   int id,
3   int customerId,
4   String type,
5   double balance) {}
```

Adding Methods to Records

```
1 public record Account(
2   int id,
3   int customerId,
4   String type,
5   double balance) {
6
7   public void addedMethod() {
8     //...
9   }
10
11 }
```

Working with Records

```
1 Account account = new Account(12345, 54321, "CHECKING",
  2178.5);
2
3 account.setType("SAVINGS"); //gives compilation error
4
5 double balance = account.balance();
```

Wrapper Classes

```
int number1 = 5;

Integer number2 = 5;
```

Wrapper Classes

Primitive Data Type	Wrapper Class
int	Integer
double	Double
long	Long
float	Float
boolean	Boolean
char	Char
byte	Byte



Arrays

special variables that can hold multiple values

Variable

```
int age = 21;
```

Array

```
int[] lottoTicket = new int[6];
```

Array Elements

```
int lottoTicket[ ] = new int[6];
```



Array Elements

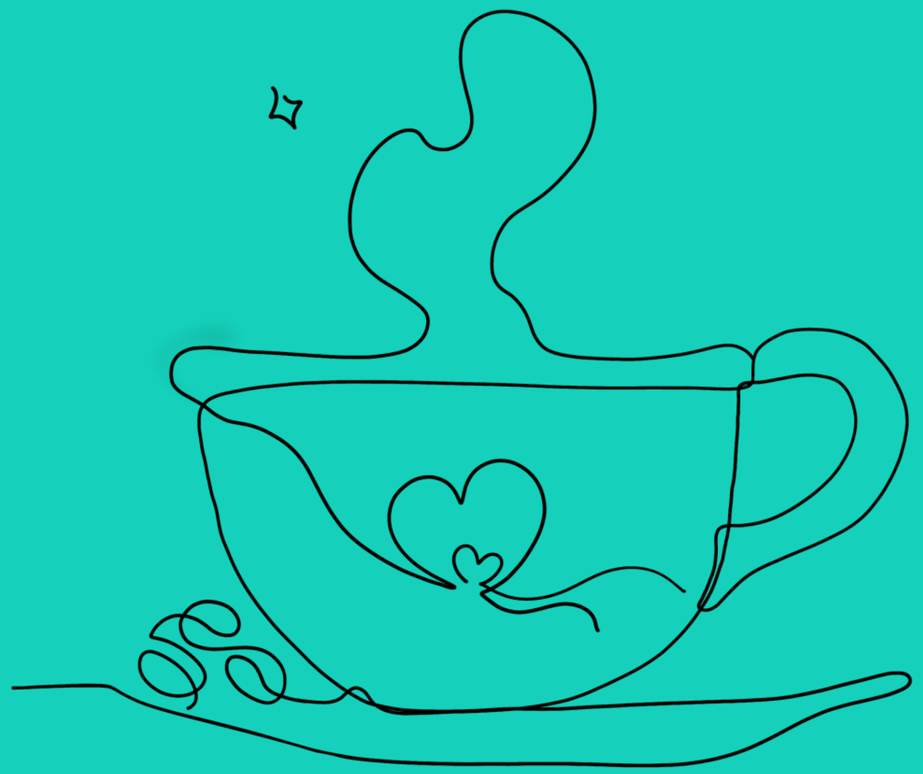
```
int[] lottoTicket = {8, 14, 22, 38, 45, 56};
```

8	14	22	38	45	56
0	1	2	3	4	5



Lottery Ticket

Create a Lottery Quick Pick application that will generate a lottery ticket with 6 random numbers between 1-69.



Strings

A sequence of characters



Word Count

Write a method that counts the number of words in a String and prints them individually on new lines



Backwards String

Write a method that prints a given String backwards. For example, if given "pots", it prints "stop".

StringBuilder

Used to create a mutable String that can be modified



Jumbled String

Write a method that adds spaces to a jumbled String where all words were written together with no spaces. Each new word begins with a capital letter.

Text Blocks

Used to work with multi-line Strings

Text Blocks

```
String response =
"[\n" +
"  {\n" +
    "    \"id\": 13344,\n" +
    "    \"customerId\": 12212,\n" +
    "    \"type\": \"CHECKING\",\n" +
    "    \"balance\": 4022.93\n" +
    "  }\n" +
    "]";
```

Text Blocks

```
1 return """
2         [
3           {
4             "id": 13344,
5             "customerId": 12212,
6             "type": "CHECKING",
7             "balance": 3821.93
8           }
9         ]
10       """;
```

Text Blocks

```
return """ Hey y'all! """;
```



Text Blocks

```
return """
        Hey y'all! """;
```



Text Blocks

```
return """
       Hey y'all!
       """;
```

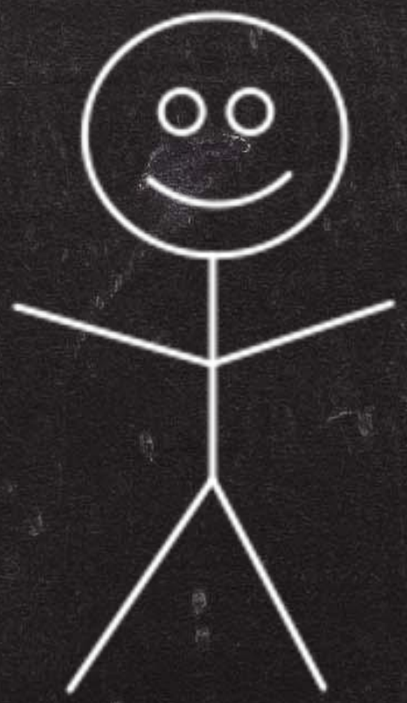


Inheritance

one class becomes an extension of another class, therefore inheriting the members of that class

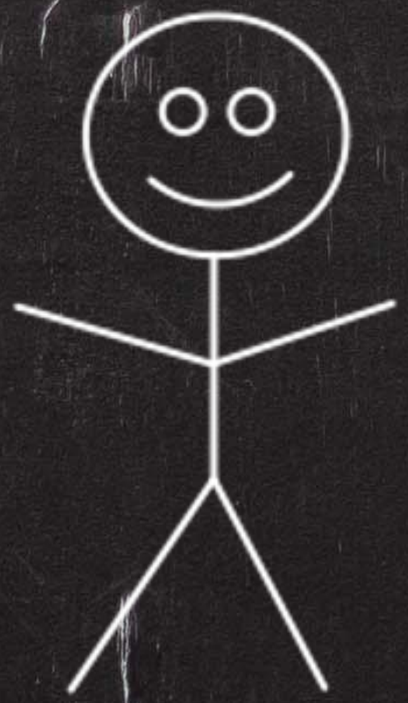
Parent and Child

Person



is a

Employee





Employee is a Person

Create a Person class, then create an Employee class that inherits from Person.

Constructors in Inheritance

constructors in superclasses execute
before constructors of the subclass



Add Constructors

Add constructors to Person and Employee classes

Constructors in Inheritance



- superclass constructors run first
- super( ) is used to call specific constructors
- super( ) calls must be first statement
- some constructor must be executed

Overriding Inherited Methods

subclasses can override the behavior
of a method it inherited



Override Perimeter

Create a Square class that inherits from Rectangle, and override its calculatePerimeter() method

Overloading Inherited Methods

subclasses can overload methods that are inherited from a superclass



Overload Inherited Method

Create a print() method in superclass then overload it in subclass

Chain of Inheritance

a class can only explicitly inherit from one superclass.

but a superclass can also inherit from another class.

```java
public class Vehicle {

    private String color;
    private int mileage;


    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public int getMileage() {
        return mileage;
    }

    public void setMileage(int mileage) {
        this.mileage = mileage;
    }
}
```

```java
public class Car extends Vehicle {
    private int doors;

    public int getDoors() {
         return doors;
     }

     public void setDoors(int doors) {
        this.doors = doors;
     }
}
```

```java
public class Coupe extends Car {

    public Coupe(){
        setDoors(2);
    }
}
```

```java
public class Garage {

    public static void main(String[] args){

        Coupe myCar = new Coupe();
        myCar.setColor("red");
    }
}
```

Limited Access in Inheritance

when a subclass inherits from a superclass,
not everything is inherited

Limited Access in Inheritance



- constructors are not inherited
- public and protected members are inherited
- private members are not inherited
- final methods can be inherited but not overridden

Limited Access in Inheritance

```java
public class Rectangle {
    protected double length;
    protected double width;
    protected double sides = 4;

    ///…
}
```

```java
public class Square extends Rectangle {
    @Override
    public double calculatePerimeter(){
        return sides * length;
    }
}
```

Access Modifiers for Overridden Methods

```java
public class Rectangle {

    public double calculatePerimeter(){
        return (2 * length) + (2 * width);
    }
}
```

```java
public class Square extends Rectangle {

    @Override
    public double calculatePerimeter(){
        return sides * length;
    }
}
```

Access Modifiers for Overridden Methods

```
public class Rectangle {

    public double calculatePerimeter(){
        return (2 * length) + (2 * width);
    }
}
```

```
public class Square extends Rectangle {

    @Override
    protected double calculatePerimeter(){
        return sides * length;
    }
}
```

Access Modifiers for Overridden Methods

```java
public class Rectangle {

    protected double calculatePerimeter(){
        return (2 * length) + (2 * width);
    }
}
```

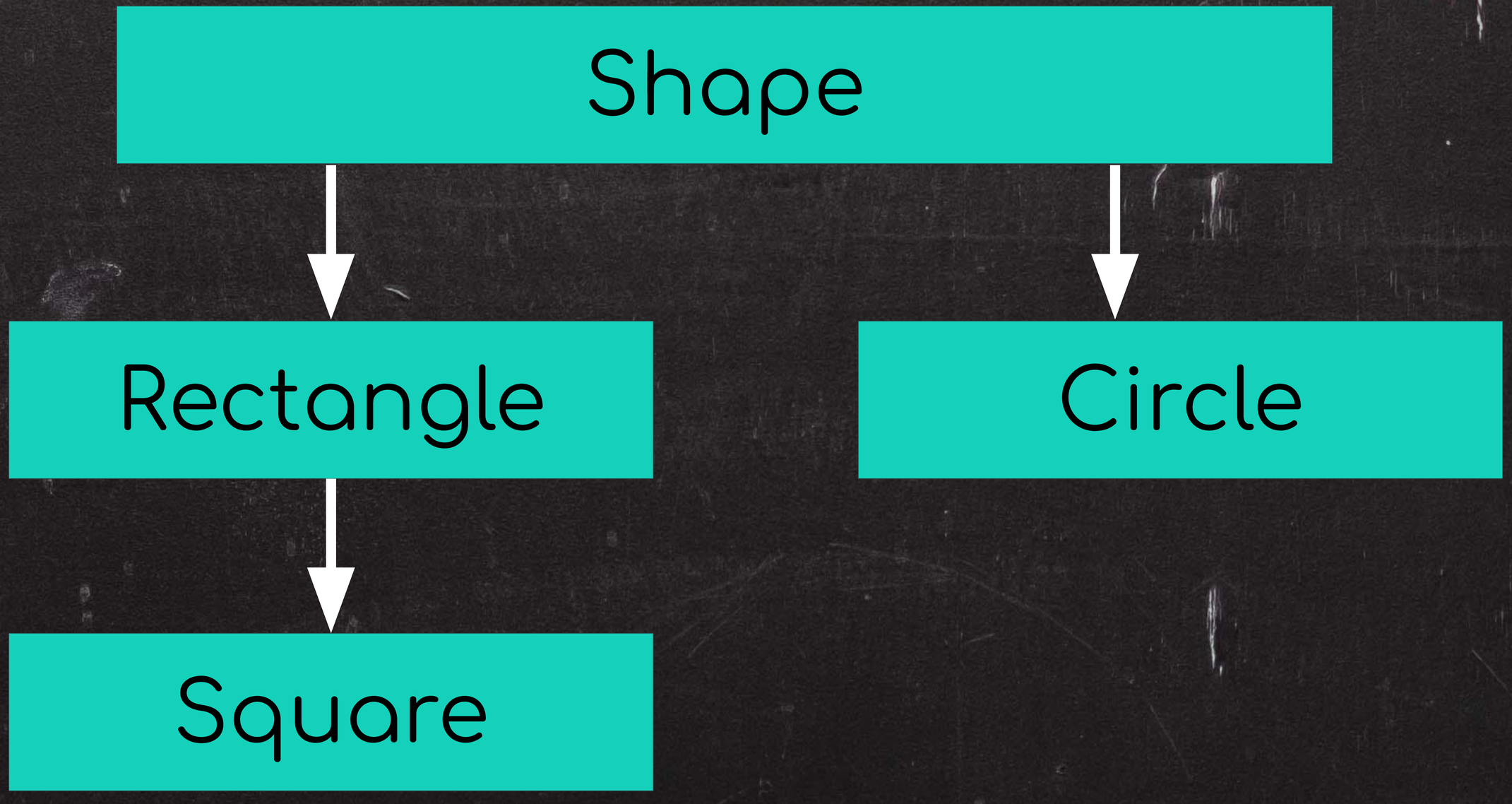
```java
public class Square extends Rectangle {

    @Override
    public double calculatePerimeter(){
        return sides * length;
    }
}
```

Sealed Classes

classes that restrict inheritance to specific classes

Sealed Classes



Sealed Classes

```
public sealed class Shape {}
```

Sealed Classes

```
public sealed class Shape permits {}
```

Sealed Classes

```
public sealed class Shape permits Rectangle, Circle {}
```

Sealed Classes

```
public sealed class Shape permits Rectangle, Circle {}

public class Rectangle extends Shape {}

public class Circle extends Shape {}
```

Sealed Classes

```
public sealed class Shape permits Rectangle, Circle {}
public sealed class Rectangle extends Shape permits
Square {}
```

Sealed Classes

```
public sealed class Shape permits Rectangle, Circle {}

public sealed class Rectangle extends Shape permits
Square {}

public non-sealed class Circle extends Shape {}
```

Sealed Classes

```
public sealed class Shape permits Rectangle, Circle {}

public sealed class Rectangle extends Shape permits
Square {}

public non-sealed class Circle extends Shape {}

public final class Square extends Rectangle {}
```

Sealed Classes

- sealed classes grant inheritance permission

- permitted subclass must extend sealed class

- permitted subclass must be sealed, non-sealed, or final

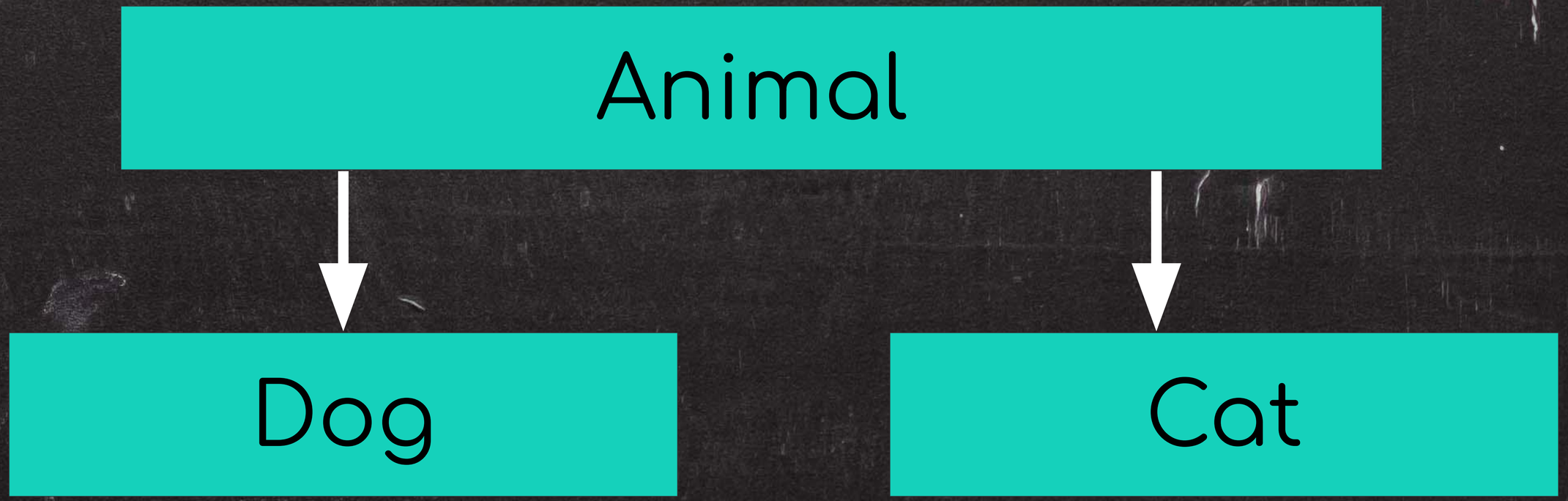




Polymorphism

ability to take multiple forms

Polymorphism



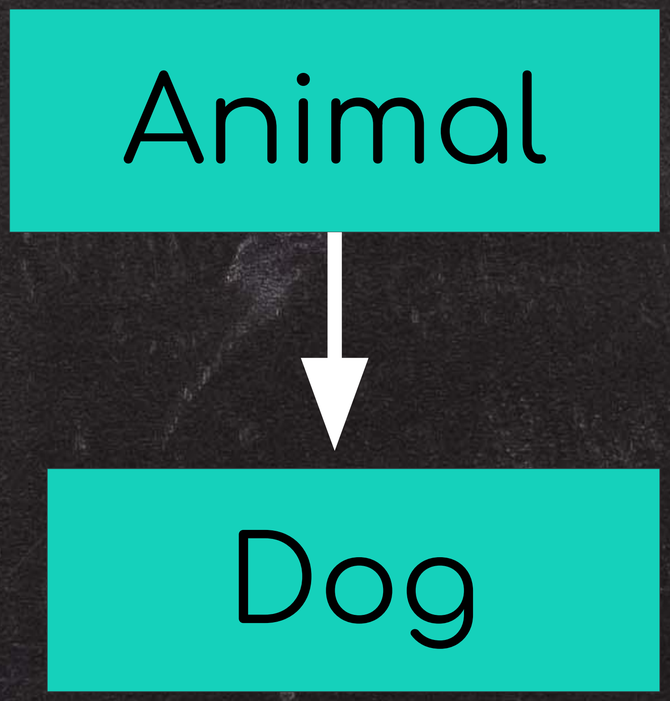
Polymorphism

```
Animal dog =
```

Polymorphism

```
Animal dog = new Dog();
```

Polymorphism




```
Animal dog = new Dog();
```



Transforming Animals

Demonstrate an example of polymorphism between an Animal, Dog, and Cat

Polymorphism





Type vs. Instance

an object can have a superclass type and a subclass instance



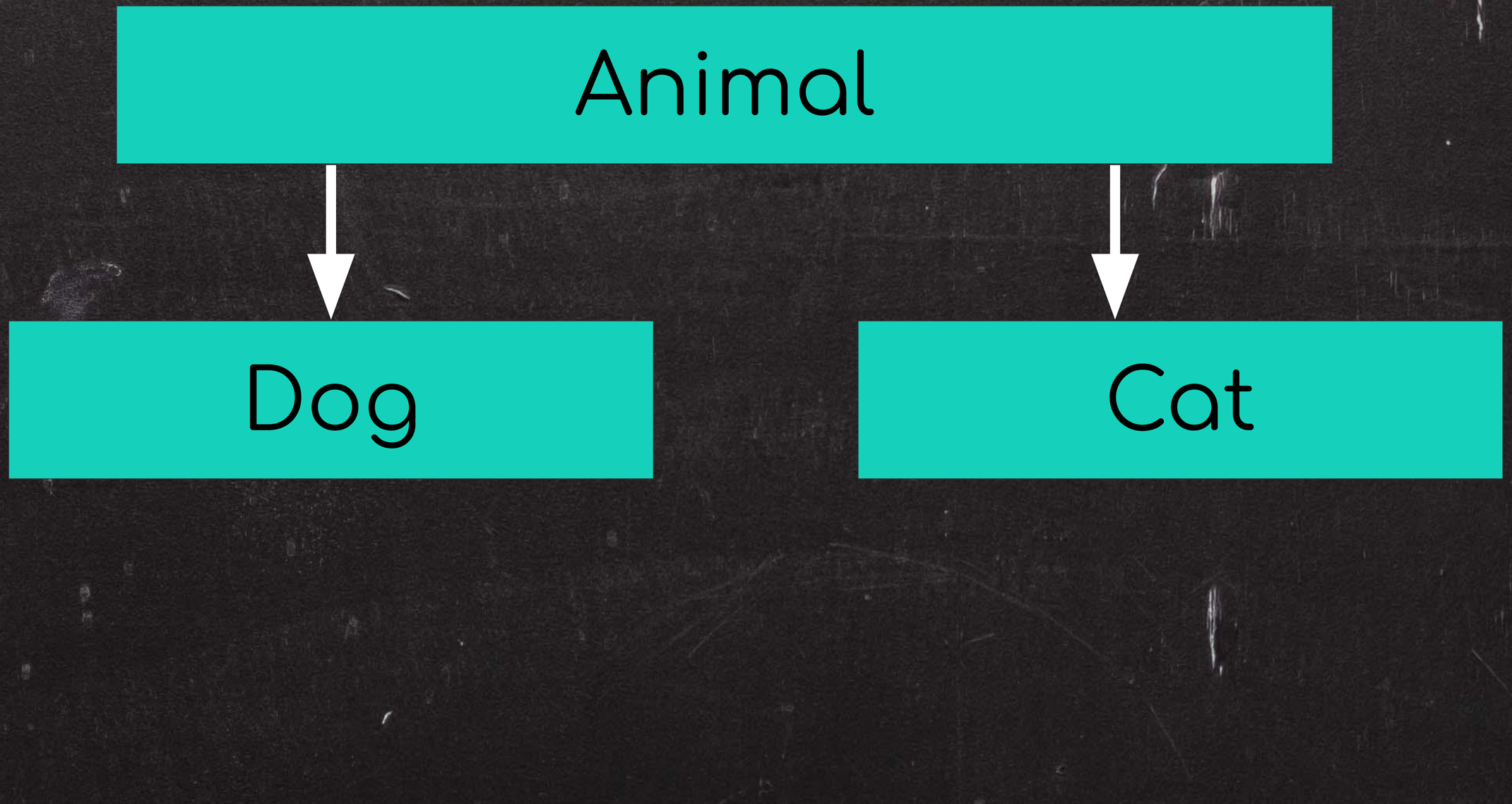
Overridden Methods

a polymorphic object will execute its overriding method at runtime

Type Casting

the action of converting an object's type into a different type

Type Casting



Type Casting

```
Animal sasha = new Dog();
```

Type Casting

```java
public class Dog extends Animal {

    @Override
    public void makeSound(){
        System.out.println("woof");
    }

    public void fetch(){
        System.out.println("fetch is fun!");
    }
}
```

```java
Animal sasha = new Dog();
sasha.fetch();
```



Sasha's itch

Downcast sasha to enable her to scratch her itch.

Type Casting

```
Animal sasha = new Dog();
Dog sashaTheDog = (Dog)sasha;
```

Type Casting

```
Animal sasha = new Dog();
Dog sashaTheDog = (Dog)sasha;
```

Type Casting

```
Animal sasha = new Cat();
Dog sashaTheDog = (Dog)sasha;
```

Type Casting

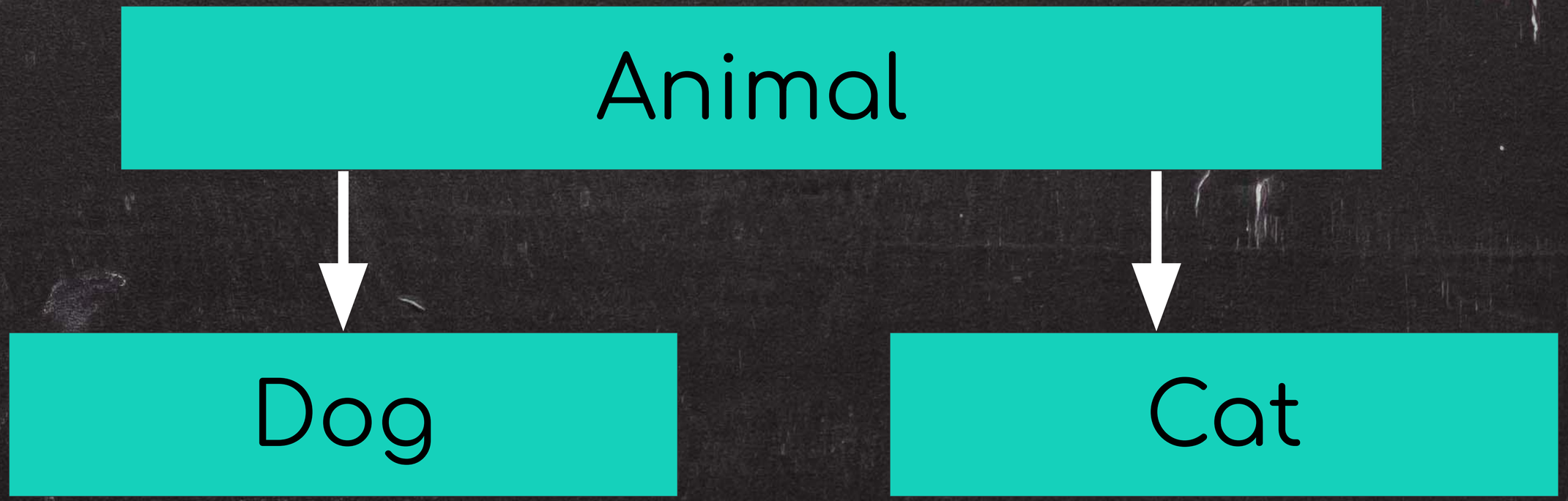
```
Animal sasha = new Cat();
Dog sashaTheDog = (Dog)sasha;
```



instanceof Operator

allows you to check if an object is an instance of a specific class

instanceof Operator



instanceof Operator

```
Animal sasha = new Dog();
```

instanceof Operator

```
Animal sasha = new Dog();
//returns true
boolean isAnimal = sasha instanceof Animal;
```

instanceof Operator

```
Animal sasha = new Dog();
//returns true
boolean isDog = sasha instanceof Dog;
```

instanceof Operator

```
Animal sasha = new Dog();
//returns false
boolean isCat = sasha instanceof Cat;
```



Feed the Animals

Create a method that feeds any animal the type of food it eats

instanceof Pattern Matching

```
Animal sasha = new Dog();

if(sasha instanceof Dog sashaTheDog) {
    sashaTheDog.fetch();
}
```

instanceof Pattern Matching

```
Animal sasha = new Dog();

if(sasha instanceof Dog sashaTheDog) {
    sashaTheDog.fetch();
}

//same as:
if(sasha instanceof Dog){
    Dog sashaTheDog = (Dog)sasha;
    sashaTheDog.fetch();
}
```



Abstraction

structures that represent abstract ideas

Abstraction

```
public abstract class Shape {

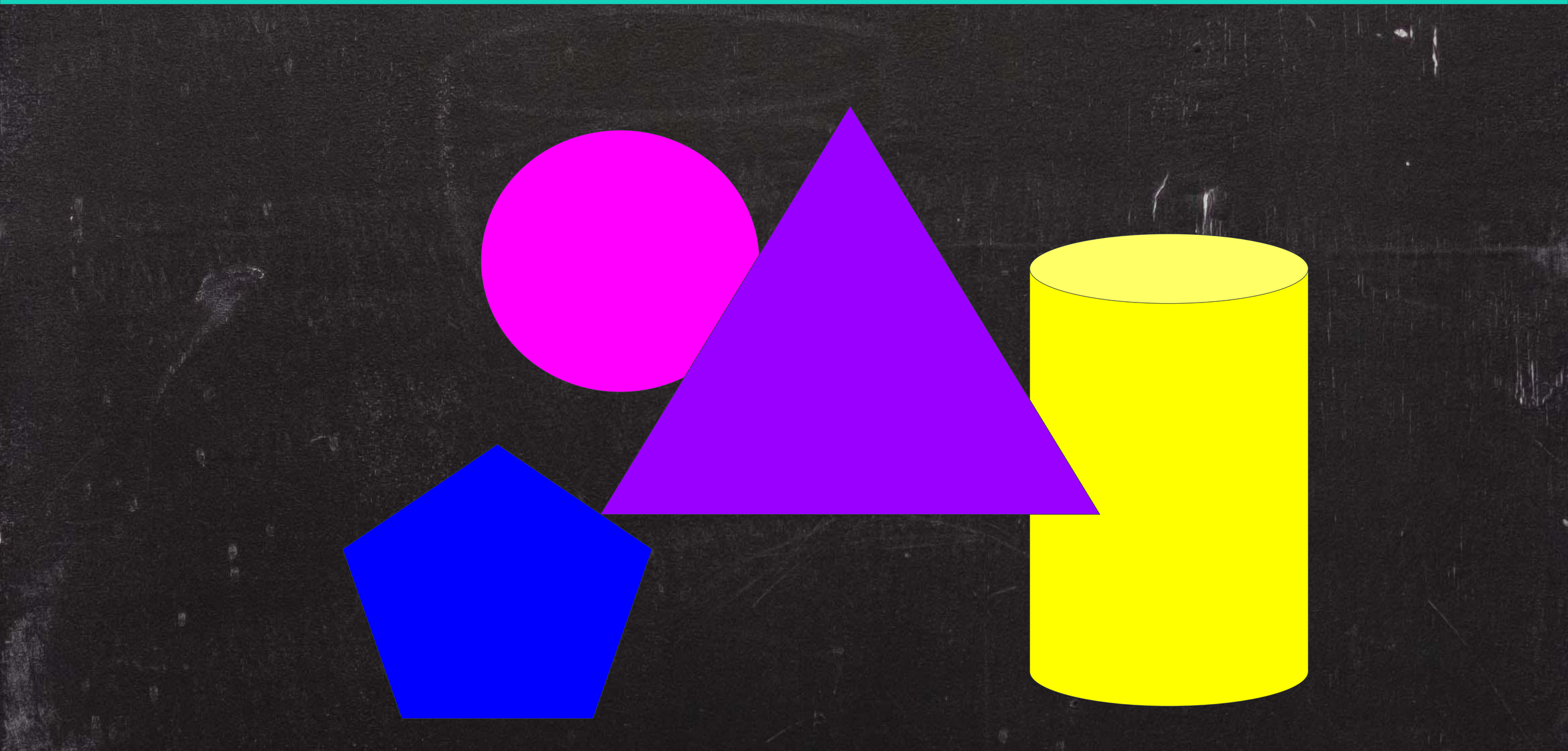
}
```



Designed to be implemented

Abstract classes are implemented by their subclasses

Abstraction



Abstraction

```
public abstract class Shape {

}
```

Abstraction

```
public abstract class Shape {
    abstract double calculateArea();

}
```



Implement a shape

Create an abstract Shape class and create a Rectangle class that inherits from it.

Abstraction

- abstract classes and methods are templates

- declare using *abstract* reserved word

- subclasses must implement inherited abstract methods

- abstract classes can't be instantiated



Interfaces

a stateless construct
with abstract
behaviors



Implement an Interface

Create a Product interface.

Create a Book class that implements Product.

Multiple Inheritance

classes can implement multiple interfaces
to achieve multiple inheritance

Multiple Inheritance

```
public class ElectronicBook implements Product, DigitalFile
{


}
```

Multiple Inheritance

```
public class ElectronicBook implements Product, DigitalFile
{
    //implement all abstract methods from Product

    //implement all abstract methods from DigitalFile

}
```

Multiple Inheritance

```
interface Product { String getSize(); }
interface DigitalFile { String getSize(); }

public class ElectronicBook implements Product, DigitalFile
{
    public String getSize() {
      //implementation
    }
}
```

Multiple Inheritance

```
interface Product { String getSize(); }
interface DigitalFile { byte getSize(); }

public class ElectronicBook implements Product, DigitalFile
{
    public String getSize() {
      //implementation
    }
}
```

Types of Methods in Interfaces



- abstract
- default
- static



Product Price

Add methods to Product interface to get and set the price

Static Methods

interface methods that contain implementation but are not inherited by implementing classes

Satic Methods

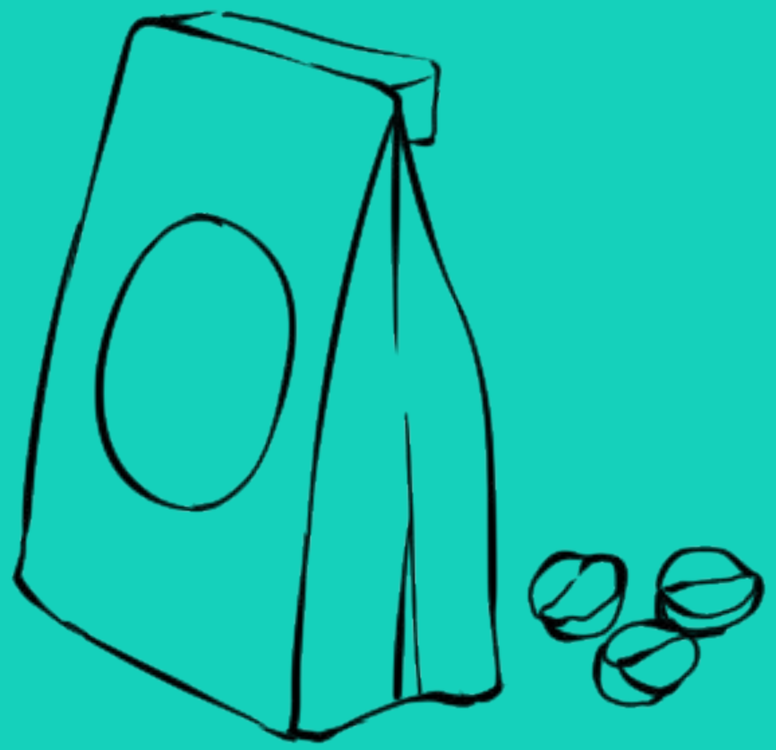
```
public static void main(String[] args) {
    Book.myStaticMethod();

    Product.myStaticMethod();
}
```

Interfaces

- can't be instantiated
- implemented by classes, extended by other interfaces
- implementing non-abstract class must implement all abstract methods
- methods are public and abstract by default
- default and static methods allowed





Data Structures

stores and organizes
a collection of objects

Collections Framework

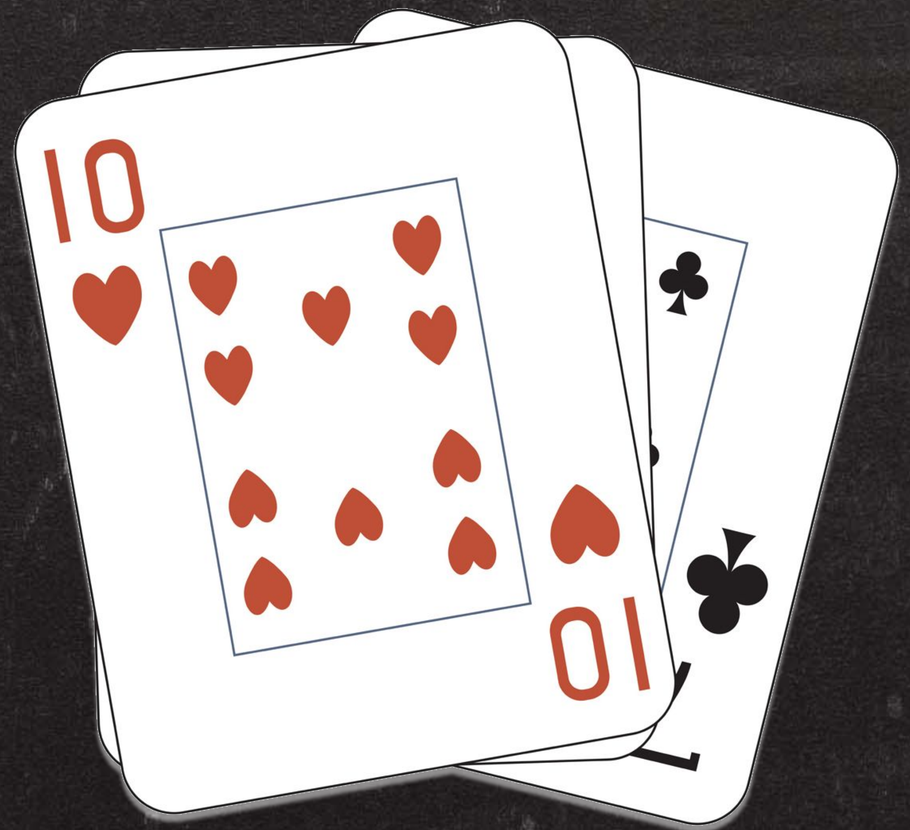
unified architecture that provides
data structures and
algorithms to manipulate them

Common Collections

- Set
- List
- Queue
- Map



Set



unordered collection
of unique objects



Fruit as a Set

Create a Set of fruit

List



ordered elements
accessible by
position



Fruit as a List

Create a List of fruit

Queue



ordered elements
that are processed
in the order in which
they are added



Fruit as a Queue

Create a Queue of fruit

Map



unordered unique
key-value pairs



Fruit as a Map

Create a Map of fruit

Iterators

object used to loop through collections



Fruit Loop

Use an iterator to loop through the Set of fruit

Access to Iterator



- Set ✓
- List ✓
- Queue ✓
- Map ✗

entrySet()

```
public static void mapDemo() {
    Set mapEntries = fruitCalories.entrySet();
}
```

entrySet()

```
public static void mapDemo() {

    Set mapEntries = fruitCalories.entrySet();
    var i = mapEntries.iterator();

    while(i.hasNext()) {
        Map.Entry entry = (Map.Entry)i.next();
        System.out.println(entry.getKey());
        System.out.println(entry.getValue());
    }
}
```

Enhanced For Loop

repetition structure designed to loop through arrays and collections



Enhanced Fruit Loop

Use Enhanced For loop to iterate over List



Enhanced Fruit Loop (Map)

Use Enhanced For loop to iterate over Map

forEach()

built-in convenience method to iterate over collections



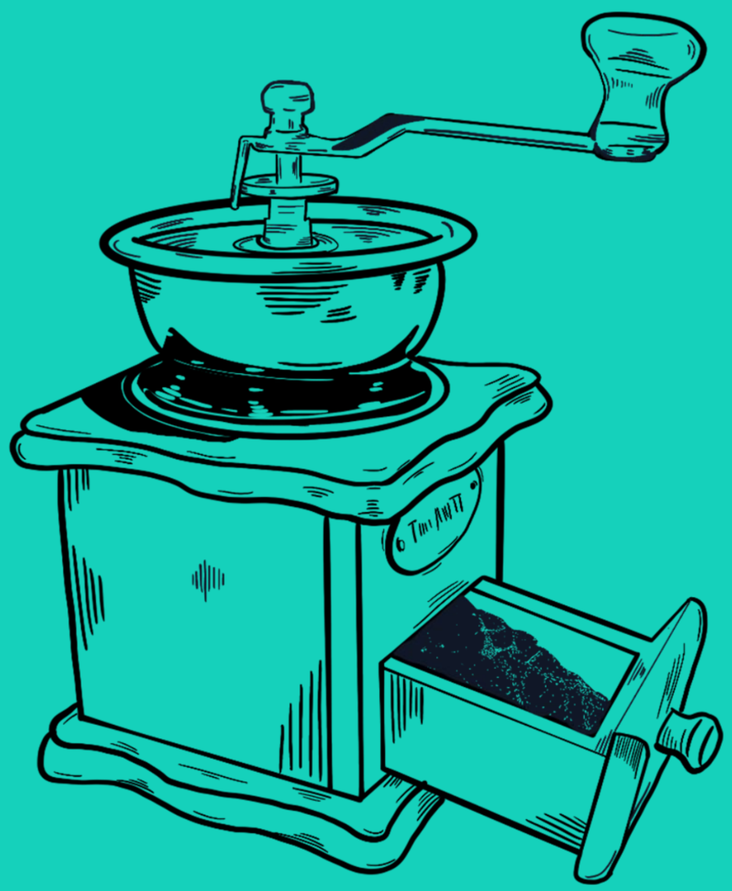
forEach Fruit

Use forEach to print elements of the fruit list



forEach Fruit (Map)

Use forEach to print elements of the Fruit Map



Functional Programming

programming style that applies pure functions to an input set to produce an output

Functional Interfaces (java.util.function)

- BiConsumer
- BiFunction
- BinaryOperator
- BiPredicate
- BooleanSupplier
- Consumer
- DoubleBinaryOperator
- DoubleConsumer
- DoubleFunction
- DoublePredicate
- DoublePredicate
- DoubleSupplier
- DoubleToIntFunction
- DoubleToLongFunction
- DoubleUnaryOperator
- Function<T,R>
- IntBinaryOperator
- IntConsumer
- IntFunction
- IntPredicate
- IntSupplier
- IntToDoubleFunction
- IntToLongFunction
- IntUnaryOperator
- LongBinaryOperator
- LongConsumer
- LongFunction
- LongPredicate
- LongToDoubleFunction
- LongToIntFunction
- LongUnaryOperator
- ObjDoubleConsumer
- ObjIntConsumer
- ObjLongConsumer
- Predicate
- Supplier
- ToDoubleBiFunction
- ToDoubleFunction
- ToIntBiFunction
- ToIntFunction
- ToLongBiFunction
- ToLongFunction
- UnaryOperator

Functional Interface

interface with a single abstract method which can be implemented by lambda expressions and method references

Functional Interfaces

```
List countries = List.of("Australia", "China", "France", "Italy");
```

Functional Interfaces

```
List countries = List.of("Australia", "China", "France", “Italy");
countries.forEach(c -> System.out.println(c));
```

Functional Interfaces

```
List countries = List.of("Australia", "China", "France", “Italy");
countries.forEach(c -> System.out.println(c));
```

java.util.function.Consumer

```java
package java.util.function;

import java.util.Objects;


@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

java.util.function.Consumer

```java
package java.util.function;

import java.util.Objects;


@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

c -> System.out.println(c)

Functional Interfaces

```
List countries = List.of("Australia", "China", "France", “Italy");

Consumer print = c -> System.out.println(c);

countries.forEach(print);
```

Functional Interfaces

```
Consumer print = c -> System.out.println(c);
print.accept("hello world");
```

Functional Interfaces

```
Consumer print = System.out::println;
print.accept("hello world");
```

INTERFACE	DESCRIPTION	ABSTRACT METHOD
Consumer	Accepts a single input argument and returns no result	void accept(T t)
Supplier	Accepts no arguments and returns a result	T get()
Predicate	Accepts one argument and returns a boolean result	boolean test(T t)
Function	Accepts one argument and returns a result	R apply(T t)
UnaryOperator	Accepts one argument and returns a result of the same type	T apply(T t)
BinaryOperator	Accepts two arguments of the same type and returns a result of the same type	T apply(T t, T u)

Functional Interfaces

```
@FunctionalInterface
public interface Function<T, R> {

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }
}
```

Functional Interfaces

```
Function<Integer, Integer> square = (n) -> n * n;
Function<Integer, Integer> add    = (n) -> n + n;
```

Functional Interfaces

```
Function<Integer, Integer> square = (n) -> n * n;
Function<Integer, Integer> add    = (n) -> n + n;

square.andThen(add).apply(5); //50
```

Functional Interfaces

```
@FunctionalInterface
public interface MyInterface {

    int add(int num1, int num2, int num3);
}
```

Functional Interfaces

```
@FunctionalInterface
public interface MyInterface {

    int add(int num1, int num2, int num3);
}
```

Functional Interfaces

```
@FunctionalInterface
public interface MyInterface {
    int add(int num1, int num2, int num3);
}
```



Streams

sequence of elements
from a data source

Stream API



Streams

```
int[] numbers = {0, 2, 4, 6, 8, 10};
```

Streams

```
int[] numbers = {0, 2, 4, 6, 8, 10};
Arrays.stream(numbers)
```

Streams

```
int[] numbers = {0, 2, 4, 6, 8, 10};

Arrays.stream(numbers)
        .forEach(n-> System.out.print(n + 1 + " "));
```

Streams

```
int[] numbers = {0, 2, 4, 6, 8, 10};
Arrays.stream(numbers)
        .forEach(n-> System.out.print(n + 1 + " "));
```

```
1   3   5   7   9   11
```

Streams

```
int[] numbers = {0, 2, 4, 6, 8, 10};

Arrays.stream(numbers).parallel()
        .forEach(n-> System.out.print(n + 1 + " "));
```

Streams

```
int[] numbers = {0, 2, 4, 6, 8, 10};
Arrays.stream(numbers).parallel()
        .forEach(n-> System.out.print(n + 1 + " "));
```

```
7   11   9   3   1   5
```

Stream Operations

Intermediate
perform operation and return resulting stream

Terminal
return a result and close the stream



Stream Operations

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");
```

anyMatch Operation

```
veggies.stream().anyMatch()
```

anyMatch Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().anyMatch(v->v.contains(" "));
```

anyMatch Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().anyMatch(v->v.contains(" "));
```

true

allMatch Operation

```
veggies.stream().allMatch()
```

allMatch Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().allMatch(v->v.contains("s"));
```

allMatch Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().allMatch(v->v.contains("s"));
```

false

filter Operation

```
veggies.stream().filter()
```

filter Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().filter(v->v.startsWith("c"))
```

filter Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().filter(v->v.startsWith("c"))
                .forEach(System.out::println);
```

filter Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().filter(v->v.startsWith("c"))
                .forEach(System.out::println);
```

```
cabbage
carrots
```

map Operation

```
veggies.stream().map()
```

map Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().map(String::toUpperCase)
```

map Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream()
                .map(String::toUpperCase)
                .forEach(System.out::println);
```

map Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream()
                .map(String::toUpperCase)
                .forEach(System.out::println);
```

```
SPINACH
CABBAGE
PEAS
GREEN BEANS
BRUSSEL SPROUTS
CARROTS
```

reduce Operation

```
List<Integer> numbers = List.of(2, 4, 6, 8, 10);

numbers.stream().reduce();
```

reduce Operation

```
List<Integer> numbers = List.of(2, 4, 6, 8, 10);
numbers.stream().reduce(0, );
```

reduce Operation

```
List<Integer> numbers = List.of(2, 4, 6, 8, 10);
numbers.stream().reduce(0, (a,b)->a+b));
```

reduce Operation

```
List<Integer> numbers = List.of(2, 4, 6, 8, 10);
numbers.stream().reduce(0, (a,b)->a+b));
```

30

reduce Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().sorted()
```

reduce Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream().sorted()
                .reduce("",(a, b) -> a + " | " + b))
```

reduce Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");
veggies.stream().sorted()
                .reduce("",(a, b) -> a + " | " + b))
```

spinach|cabbage|peas|green beans| brussel sprouts|carrots

collect Operation

```
veggies.stream().collect()
```

collect Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream()
                .filter(v->v.endsWith("s"))
                .collect();
```

collect Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

veggies.stream()
                .filter(v->v.endsWith("s"))
                .collect(Collectors.toList());
```

collect Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

List veggiesThatEndWithS = veggies.stream()
                .filter(v->v.endsWith("s"))
                .collect(Collectors.toList());
```

collect Operation

```
List<String> veggies = List.of(
        "spinach",
        "cabbage",
        "peas",
        "green beans",
        "brussel sprouts",
        "carrots");

List veggiesThatEndWithS = veggies.stream()
                .filter(v->v.endsWith("s"))
                .collect(Collectors.toList());

veggiesThatEndWithS.forEach(System.out::println);
```

Stream Operations

```
veggies.stream()
        .sorted()
        .filter(v->v.startsWith("c"))
        .map(String::toUpperCase)
        .map(v->v.transform(w->"yummy " + w))
        .forEach(System.out::println);
```

```
yummy CABBAGE
yummy CARROTS
```



Exceptions

error events that disrupt the normal flow of a program

Exceptions

```
String[] places = {“work”, “school”, “the cinema”};
for(int i=0; i<4; i++){
    System.out.println(“I am going to “ + places[i]);
}
```

Exceptions

```
String[] places = {“work”, “school”, “the cinema”};
for(int i=0; i<4; i++){
    System.out.println(“I am going to “ + places[i]);
}
```

ArrayIndexOutOfBoundsException



Create File

Write a program that creates a new file, and handles any exception that may occur.

Checked vs Unchecked

exceptions can be checked or unchecked

Checked Exception

```
public boolean createNewFile() throws IOException {

}
```

Checked Exception

```
File file = new File("resources/nonexistent.txt");
file.createNewFile();
```



Unchecked Exception

```
public double nextDouble() { }
```

Unchecked Exception

```
Scanner console = new Scanner(System.in);
double input = console.nextDouble();
```

InputMismatchException

Checked Exceptions

exceptions that are checked by the compiler

Unchecked Exceptions

exceptions that are not checked by the compiler

Checked vs Unchecked

- Check exceptions are used when there's a possibility of recovery

- Unchecked exceptions are used when there's not anything that can be done in the event of the error



Handling Multiple Exceptions



- Polymorphism
- Multiple catch blocks
- Catch multiple in single block

Polymorphism

Class ArrayIndexOutOfBoundsException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IndexOutOfBoundsException
                    java.lang.ArrayIndexOutOfBoundsException

All Implemented Interfaces:

`Serializable`

Polymorphism

```
public static void main(String args[]){
    File file = new File("resources/nonexistent.txt");
    try{
       file.createNewFile();
    } catch(IOException e){
       System.out.println(“Sorry, an error has occurred.”);
    }
}
```

Polymorphism

```
public static void main(String args[]){
    File file = new File("resources/nonexistent.txt");
    try{
        file.createNewFile();
    } catch(Exception e){
        System.out.println("Sorry, an error has occurred.");
    }
}
```

Multiple Catch Blocks

handle different exceptions with multiple catch blocks



Read File

Write a program that reads decimal numbers from a file, and handle the two types of exceptions that come from that

Catch Multiple with Single Block

handle different exceptions with
one catch block

Catch Multiple with Single Block

```
try{
    Scanner fileReader = new Scanner(file);
    while(fileReader.hasNext()){
        System.out.println(fileReader.nextDouble());
    }
}catch(FileNotFoundException e){
    e.printStackTrace();
}catch(InputMismatchException e){
    e.printStackTrace();
}
```

Catch Multiple with Single Block

```
try{
    Scanner fileReader = new Scanner(file);
    while(fileReader.hasNext()){
        System.out.println(fileReader.nextDouble());
    }
}catch(FileNotFoundException | InputMismatchException e){
    e.printStackTrace();
}
```



Close the File Reader

Update the program to close the file reader

Flow of a Program

1. Open resource

2. Read data from resource → Exception

3. Close resource

Finally Block

```
try{
    fileReader = new Scanner(file);
    while(fileReader.hasNext()){
        System.out.println(fileReader.nextDouble());
    }
}catch(FileNotFoundException | InputMismatchException e){
    e.printStackTrace();
}finally{
    fileReader.close();
}
```

Try with Resources

declaration of resources that should be automatically closed after the execution of a try block



Auto-close File Reader

Automatically close file reader within try block



Print to File

Change printer from System to PrintWriter. Auto-close both resources within try block



Throw Exception

Write a method that calculates an employee's pay but does not allow overtime as input.



Rethrow Exception

Write a program that creates a new file but doesnt' catch the exceptions



Java Fundamentals

Angie Jones

Certified Java Programmer and Java Champion