Java Fundamentals Cheatsheet

A Java cheatsheet designed to provide a quick and easy-to-use resource for developers working with the Java programming language. It includes important syntax, keywords, and concepts in a concise format that can be easily scanned and understood. It covers a wide range of topics, including data types, arrays, strings, classes, objects, methods, inheritance, interfaces, and exceptions.

Contents

1. Introduction
2. Variables
3. Booleans and Conditionals
4. Functions
5. Loops
6. Arrays
7. Objects
8. List Collections
9. Exception Handling
10. Map Collections
11. Inheritance
12. Higher Order Functions
13. Miscellaneous Topics

Introduction

- Java file names should be CamelCase format
- They should end with `.java` extension
- All of the code in java is written inside a class
- Every line of code needs a semicolon
- Java is case sensitive

Class

- Every class should have the `class` keyword and a class name like `HelloWorld`
- Class name should always match the filename it's in

```
class HelloWorld {
  // all the code
}
```

- Classes need a `main` method which acts as an entry point to the class

```
public class HelloWorld {
  public static void main(String[] args) {
    // code
  }
}
```

Running the Code

Once we run the code, Java looks for the entry point which is the main function. Once it finds the entry point, it starts executing whatever code is inside it.

If there are any errors Java won't run the code and throw an error displaying the problem and the line.

Terminal Commands

1. **javac** (Java Compiler) - compiles our code into byte code

   ```
   javac <FileName>.java
   ```

   It will compile all the code into *Bytecode* which will live inside *.class* file - `HelloWorld.class`

2. **java** executes the compiled code

   ```
   java <FileName>
   ```

Always recompile code after making any changes.

Print Statement

- `println(msg)` method is used to print msgs to console

```
System.out.println("Hello World");
```

`print` VS `println`

- `println` prints text and moves to a new line

  ```
   System.out.println(" a ");
  System.out.println(" b ");
  System.out.println(" c ");
  ```

  ```
   >> a
  >> b
  >> c
  ```

- `print` prints text but it does not move to a new line. So, ensuing print calls print on the same line

  ```
   System.out.println(" a ");
  System.out.println(" b ");
  System.out.println(" c ");
  ```

  ```
   >> a b c
  ```

Comments

Comments are written using **double slashes (`//`)**. These lines are completely ignored while compiling the code.

```
 public class Comments {
  public static void main(String[] args) {
    // this line will be ignored
    System.out.println("Hello World");
  }
}

-> Hello World
```

Variables

Variables are used for storing data.

```
 int apples = 6; // int ->  stores integers
long population = 800000000L; // long ->  stores long integer
double price = 5.99; // double -> stores decimals
String greeting = "Hello"; // String -> stores texts
char grade = 'A'; // char -> stores single characters
```

- Always choose the data type which occupies the least space out of all the suitable ones.
- Name variables using **lowerCamelCase**.
- Any variable value can be updated after it's defined as long as it follows the variable range

```
 public class Bus {
  public static void main(String[] args) {
    int passengers = 0;

    passengers = passengers + 9; // more passengers come on bus stop
    System.out.println(passengers); // 9
  }
}
```

Variable Types

int

- Uses **4 bytes (32 bits)** of memory
- Can store any value in 2 billion range
- Highest value it can store is *2,147,483,647*

```
 int apples = 6;
apples += 4;
System.out.println(apples) // 10
```

String

- Used to store text
- Unlike *int* varies in memory
- Empty text takes *24 bytes* and the more text we add more memory it takes
- Create string value using **double quotes ("some text")**
- Two strings or integers can be concatenated using the **+** operator

```
 String name = "Deekayy";
int amount = 4;
String item = "bats"
String statement = name + " has " + amount + " " + item
System.out.println(statement) // Deekayy has 4 bats
```

char

- Used to store only single characters
- Java only allocates **2 bytes (16 bits)** for char
- No matter what character we store, it only takes 2 bytes in memory
- Char values should be wrapped in **single quotes ('A')**
- `char letter = 'F'` vs `String letter = "F"` can be a diff b/w *2 bytes* vs *48 bytes*

```
 char mathGrade = 'A';
char scienceGrade = 'C';

System.out.println("Maths - " + mathGrade); // Maths - A
System.out.println("Science - " + scienceGrade); // Science - C
```

long

- Used to store large whole numbers
- Takes **8 bytes (64 bits)** of memory
- Use them if you expect the variable to cross 2 billion range (for eg, YouTube views)
- Need to put an **L** at the end of variable value to let Java know it's going to be a long integer

```
 long dailyGoogleSearches = 8000000000L;
long population = 95_000_000_000L; // makes it easier to read

System.out.println("Population is " + population); // Population is 95000000000L
```

> FAQ
>
> Ques: If I have specifically declared that a variable is to be of type long, why do I need to state this again with the "L" suffix on the value?
>
> Ans: The data type declaration and the number literal are interpreted separately, so the compiler can't infer the data type when considering the number literal.

double

- Used to store decimals
- Takes **8 bytes (64 bits)** of memory
- Can take up to 15 decimal places

```
 double gradePoint = 8.46;

System.out.println("CGPA - " + gradePoint); // CGPA - 8.46
```

> Note!
>
> Always use double instead of int for math calculations. Otherwise, 20/3 which is 6.67 (double) will be turned into 6 (int) and result in falsy calculations.

Math Operations

Always wrap maths operations in brackets when they are performed within a string.

```
 int a = 5;
int b = 7;

// wrong way
System.out.println("Sum will be " + a + b + " apples"); // ⍰ Sum will be 57 apples

// right way
System.out.println("Sum will be " + (a + b) + " apples"); // ✔ Sum will be 12 apples
```

This is done to provide some kind of order for things to happen.

The mathematical operators in Java are -

- + (plus)
- - (minus)
- * (multiplication)
- / (division)
- % (modulus - returns the remainder)
- ++ (add 1)
- -- (subtract 1)
- += (add by - increase value by number on the right)
- -= (subtract by - decreases value by number on right)

Type Casting

Variable values can be casted from one type to another. It can be done by using a parenthesis before the variable and putting the required type in that parenthesis.

```
 int a = 20;
int b = 3;

System.out.println(a / b) // 6
System.out.println((double) a / b) // 6.666...
```

Scanner

Contains methods that can be used to scan for user input.

- `nextInt()` - used to scan Integers
- `nextDouble()` - used to scan Decimals
- `nextLine()` - used to scan Text

Scanner with all of its logic and methods is defined inside *Java.utils* and therefore we need to import it first.

```
 import java.util.Scanner;

// create a new scanner instance
Scanner scanner = new Scanner(System.in);

// use scanner methods to get user data
System.out.println("What's your name?");
String name = scanner.nextLine();

System.out.println("What's your age?");
int age = scanner.nextInt();

// close the scanner instance
scanner.close()
```

Also, always close the scanner instance when you are done with it to avoid any memory leak.

The first `nextLine()` after `nextInt()`, `nextLong()`, or `nextDouble()` gets skipped and the solution is to put a throwaway or temporary `nextLine()` in between which will be ignored.

> There are a bunch of escape characters that can be used to special things.
>
> **\n** can be used to add a new line or line break whereas **\t** is used for adding a tab space.

Delimiters

- White space that separates input data
- The scanner methods like **nextInt()**, **nextLong()**, **nextDouble()**, and **next()** skip the delimiter value
- Delimiter values are the white space between multiple user inputs. For eg, if a user enter `20 45 234`, the empty space in between are the delimiters.
- `nextInt()` method ignores these empty spaces and only captures the next integer values

scan.nextLine()

- It reads everything line by line
- unlike the above methods, this reads all the user inputs as one string text and doesn't ignore any white space
- For eg, if a user enters -

```
I am Batman
```

- `next()` method will be called 3 times for each of the three words since they are separated by white spaces and hence we'll get 3 different string values - "I", "am", and "Batman"
- `nextLine()` method will be used only once since it won't ignore the spaces in between and we'll get one single string - "I am Batman"

`scan.nextLine()` trap

We have to use a throwaway scan.nextLine() method after the other scanner methods since when we read values for other methods from the user inputs, the line doesn't end.

Since `.nextLine()` doesn't ignore the spaces it starts from the same place we read the last integer on and hence the first `.nextLine()` captures the remaining line of the other scanner methods and the next `.nextLine()` is used for our purpose.

For eg: User enters an integer 46 and then enters a string as follows -

```
 46 <EMPTY SPACE>
Hello John
```

Here `.nextInt()` will read 46 and `nextLine()` will read that remaining empty space on that line therefore we'll need to add an extra `nextLine()`.

If we used `next()` instead of `nextLine()` after `nextInt()` it would have ignored the empty space and moved to the next line. But then it would have only captured "Hello" and not "John" since it doesn't capture empty spaces.

Booleans & Conditionals

A Boolean is only limited to two values - **true or false**. Java only allocates the smallest unit to it which is **1 Bit**.

```
 boolean b1 = true;
boolean b2 = false;

System.out.println(b1); // true
System.out.println(b2); // false
```

They are most useful in the form of comparisons.

> Note: `true` and `false` are always lowercased.

Comparisons

There are 8 types of comparison operators.

Operator	Comparison
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to

Operator	Comparison
equals	equal to (for String)
!equals	not equal to (for String)

`equals` method for String works like

```
String.equals(String)
```

Key Points to Remember

1. Never use `==` or `!=` to compare String values. It produces weird results.
2. Don't confuse `=` and `==`.
   - `=` sets a variable equal to a new value
   - `==` compares 2 values and returns a boolean

If-Else Statements

- Runs code only if a condition is true
- A comparison within a if statement is called a *condition*

```
 int grade = 65;

if (grade > 50) {
System.out.println("You passed!");
}
```

- When if condition is false, Java runs the *else* statement

```
 int grade = 65;

if (grade > 80) { // false
   System.out.println("You topped!");
} else {
   System.out.println("You passed but didn't top!");
}
```

> We need to be careful while putting the conditions, i.e., keep `greater than and equal` to in mind too.

Nested If-Else Statements

- Between if and else, more if-else statements can be added to test many conditions.
- Java only runs the first statement that ends up being true and ignores the rest.

```
 int grade = 75;

if (grade >= 90) {
   System.out.println("You got an A+ grade!");
} else if (grade >= 80) {
   System.out.println("You got an A grade!");
} else {
   System.out.println("You failed and got an F grade!");
}
```

Logical Operators

- Can connect many comparisons into one condition.
- Most common ones are **OR** and **AND**.

OR (`||`)

- returns true if any one comparison is true

```
int chemGrade = 65;
int engGrade = 75;

if (chemGrade > 70 || engGrade > 70) { // true
System.out.println("You got a scholarship!!!"); // runs
} else {
System.out.println("Haha noob");
}
```

AND (`&&`)

- returns true if any all of the comparisons are true
- if any one of the condition is false, entire comparison becomes false

```
int credits = 52;
double GPA = 3.45;

if (credits >= 50 && GPA > 3.5) { // false
System.out.println("You got a diploma!!!");
} else {
System.out.println("No diploma"); // runs
}
```

Switch Statements

- Compares a value against a list of cases
- When the value matches to one of the given cases then the code for that particular case is executed
- If values doesn't match any of the cases, the default case code runs

> Note: When the value matches a case, it turns on the switch and the code of every following case is run. To avoid this we have the **break** keyword.

`Break` Keyword

- Used to break the switch statement
- When a case is met, this keyword is used to break out of the switch statements so that the rest of the cases don't get executed

```
int day = 5; // Friday

switch (day) {
case 1:
   System.out.println("Monday");
   break;
case 2:
   System.out.println("Tuesday");
   break;
case 3:
   System.out.println("Wednesday");
   break;
case 4:
   System.out.println("Thursday");
   break;
case 5:
   System.out.println("Friday");
   break;
case 6:
   System.out.println("Saturday");
   break;
case 7:
   System.out.println("Sunday");
   break;
default:
   System.out.println("Please enter a valid day...");
}
```

The above code will output -

```
>> Friday
```

However, if we didn't add break statement, the output would be -

```
>> Friday
>> Saturday
>> Sunday
>> Please enter a valid day...
```

`Switch` VS `If-Else`

- If-Else are more suitable with conditions and ranges where as switch is better when there are a finite no. of cases to compare with one value.
- If there are multiple conditions if-else if more suitable
- 90-95% of times, if-else will be used

if-else runs a code if a condition is *true* **switch** runs a code if an argument matches a case

Functions

A **Function** is a group of code that performs a particular task. Functions make your code more organized and reusable.

There are 5 components for every function -

1. Level of access
2. Return value
3. Function name
4. Parameters
5. Code inside functions

```
// 1     2    3      4
public void name (int param) {
  // code -> 5
}
```

- Function can be called as many times as needed using syntax `functionName()`
- Function names should be *lowerCamelCase*
- Functions are also called methods at times

Parameters

- Stores a value that functions expect to receive
- Function with parameters expect values to be passed to function during function call
- For every parameter we need to pass one value to the function

Arguments

- Values that we pass into function
- These are the actual values that will be used inside function

```
// len & bth -> parameters
public static void calculateArea(double len, double bth) {
  System.out.println("Area: " + len * bth);
}

// 10 & 20 or 9.3 & 21 -> arguments
public static void main(String[] args) {
  calculateArea(10, 20); // 200.0
  calculateArea(9.3, 21); // 195.3
}
```

Return Statements

- Functions can return a value at the end
- Return type must be specified at the time of function declaration
- Return type are means what type value will be returned from the function

```
// - <return_type> -
public double calcArea(double len, double bth) {
  double area = len * bth;
  return area; // <return value>
}
```

- Function call stores the returned value and later it can be stored inside a variable

```
double curArea = calcArea(10.0, 5.0);
System.out.println(curArea) // 50.0
```

- Return statement stops the function execution then and there and no further lines of code are run inside that function

Every function should always perform only one task!

- If a function is only printing something and doesn't perform any computation, it doesn't need to return anything and should have the **void** return type
- If a function is performing any computation/calculation that it should just return the calculated value and let the parent function decide what to do with the returned value

Doc Comments

- Can be used to describe what a function does to guide your code
- Key components:
  - What function does
  - Parameters and their types
  - What it returns
- Makes it easy for other collaborators to understand what our code does

```
/**
 * Function name: printText
 *
 * @param name (String)
 * @param age  (int)
 * @return msg (String)
 *
 * Inside the function:
 *  1. Build the message: "Hi, my name is <name> and I am <age> years old."
 *  2. return the message
 */

public static String printText(String name, int age) {
  String msg = "Hi, my name is " + name + " and I am " + age + " years old.";
  return msg;
}
```

Scope

Scope refers where the variable can be accessed from. They are either class scoped or function scoped.

Function Scope

A variable inside a function can only run inside the function scope. It does not exist outside the function it is declared in.

```
public static void main(String[] args) {
  System.out.println(apples); // Throws error
  someFunction();
}

public static void someFunction() {
  int apples = 5;
  System.out.println(apples); // Prints `5`
}
```

In the above code,

- Since `apples` variable is declared inside `someFunction`, it can be accessed (or printed) in the main function. We'll get an error.
- Similarly, if the variable had been declared in the `main` function, we wouldn't have been able to use it in `someFunction`.

Class Scope

A variable in a class exists in class scope. Scope of a class spans the entire code inside the class. Basically any variable declared in the class scope is accessible in anywhere inside the class.

```
public class Scope {
  static int dogs = 5;

  public static void main(String[] args) {
    System.out.println(dogs); // Prints `5`
    someFunction();
  }

  public static void someFunction() {
    System.out.println(dogs); // Prints `5`
  }
}
```

Built-In Functions

- Functions that are already made for us
- `println(arg)` is a built-in void function that just prints the argument we pass and doesn't return anything

Basically, behind the scenes somewhere in Java there is a code like

```
public void println(String x) {
  // many lines of code
}
```

- `println()` function isn't local to our class, it's attached to some other class and hence we need to use `System.out` before using it

Math Functions

There are several built-in maths functions already made in Java for us to use like `log`, `sin` or `cos`.

- Usually math functions expect a *double* parameter and return a *double* value.
- Some math functions expect more than 1 parameter. For eg, `Math.pow(2,4)` to calculate powers.
- There are 1000s of such functions and there is no need to memorize them all. **Use Google**.

```
public static void main(String[] args) {
  double sine = Math.sin(1.2);
  System.out.println("sine: " + sine); // 0.9320390859672264

  double power = calcPower(2, 4);
  System.out.println("power: " + power); // 16.0
}

public static double calcPower(double base, double exponent) {
  return Math.pow(base, exponent);
}
```

Loops

Loops are used to run a block of code multiple number of times. There are 2 types of loops -

- For loops
- While loops

Use a **for loop**:

- when you know how many times a code needs to run.

Use a **while loop**:

- when it's not clear how many times a code needs to run

For Loops

- Runs code a specific number of times

- A for loop has 3 parts -
    1. Starting point: `int i=1`
    2. Stop condition: `int i<=3`
    3. Step size: `i++`

```
// for loop with 3 iterations
for (int i = 1; i <= 3; i++) {
  System.out.println(i);
}
```

In the above code,

- **i** starts with value 1 and we run the code inside the loop and print 1
- Then we increment **i** and run the code where `i=2` and print 2
- Then we increment it again and **i** becomes 3 and we repeat the previous step
- When 3 is printed and we increment **i** again and it becomes 4
- Hence the condition `i<=3` becomes false for the next iteration
- We break out of the loop

Final Output -

```
1
2
3
```

While Loops

- Keeps running code **while** something is **true**
- Only thing a while loop needs is a **condition**

```
int num = 25;

while (num <= 30) {
  System.out.println(num);
  num++;
}
```

In the above code,

- Initially since the condition `num<=30` is true we enter the while loop
- In the first iteration **num** is printed as 25 and then we increment **num** by 1
- It becomes 26 and next iteration prints 26 and increments **num** again
- The loop continues until 30 is printed and **num** becomes 31
- Now the condition `num<=30` becomes false and we break out of the loop

Incorrect Code ❌

```
int num = 25;

while (num <= 30) {
  System.out.println(num);
}
```

In the above code,

- The condition `num <=30` will always be true
- The loop will run forever and will eventually crash the program

> In real world, we should have used a for loop for this example since we know how many times we need to run the code since we know the ending condition, i.e., when num becomes 31.

`Break` and `Continue`

The `break` and `continue` statements give us more control over our loops.

Continue

- Skips an iteration and continues to the next iteration

```
for (int i = 1; i <= 5; i++) {
  if (i % 2 == 0) {
    continue; // skips the iteration if above condition is true
  }
  System.out.println(i);
}
```

Output -

```
1
2
5
```

Break

- Breaks the loop and stops all the upcoming iterations

```
for (int i = 1; i <= 5; i++) {
  if (i % 2 == 0) {
    break; // exits the loop if above condition is true
  }
  System.out.println(i);
}
```

Output -

```
1
```

Nested Loops

- Nested loops are loops inside loops
- Have an **outer loop** and an **inner loop**

```
for (int i = 1; i <= 3; i++) {
  for (int j = 1; j <= 3; j++) {
    System.out.println("i: " + i + ", j: " + j);
  }
}
```

In the above code,

- Every iteration of outer loop will run the inner loop 3 times (From j=1 to j=3)
- Outer loop itself will also have 3 iterations (From i=1 to i=3)
- Hence the loop will run a total of 3x3 = 9 times and the output will be -

```
i: 1, j: 1
i: 1, j: 2
i: 1, j: 3
i: 2, j: 1
i: 2, j: 2
i: 2, j: 3
i: 3, j: 1
i: 3, j: 2
i: 3, j: 3
```

Applications of Nested Loops

- Useful when working with 2D arrays

Arrays

- Used to store many values at one time in an organized way
- All the values have to be of the same type
- These values are somewhat related to one another

To create an array of integers -

```
int[] integers = {1, 2, 3};
```

To create an array of strings -

```
String[] words = {"hi", "hello", "hola"};
```

To create an array of decimals -

```
double[] words = {1.2, 2.2, 3.3};
```

- Variable does not store the array directly, it store the reference that points to the created array
- Printing a whole array prints its Hashcode representation

```
String[] names = { "John", "Mary", "Bob" };
System.out.println(names); // [Ljava.lang.String;@5acf9800 (memory address)
```

- Each array element is stored at an index (starting from 0) and can be printed separately

```
                  // 0       1       2
String[] names = { "John", "Mary", "Bob" };
System.out.println(names[0]); // John
System.out.println(names[1]); // Mary
System.out.println(names[2]); // Bob
System.out.println(names[3]); // Array Index Error
```

Looping Arrays

Arrays can be looped over so we don't need to perform a single operation for every single element one by one.

The `length` property

- Returns the number of elements in an array

```
String[] names = { "John", "Mary", "Bob" };
System.out.println(names.length); // 3
```

- Used a lot when looping over arrays

```
String[] names = { "John", "Mary", "Bob" };

for (int i = 0; i < names.length; i++) {
  System.out.println(names[i]);
}
```

Tips

- Always use `length` property to count elements, never do it yourself
- Be careful while using the condition, since the last element will always have the index 1 less than the array's length
- In the above case, if we used `i <= names.length` it will throw an error since at one point `names.length` will be 3 and if we enter the loop with `i=3` we won't get any element since last element has the index 2.

Updating Arrays

- Use the element index and directly update it's value

```
String[] names = { "John", "Mary", "Bob" };
System.out.println(Arrays.toString(names)); // [John, Mary, Bob]

// Update the value at index 1 and index 2
names[1] = "Alice";
names[2] = "Charlie";
System.out.println(Arrays.toString(names)); // [John, Alice, Charlie]
```

`Arrays.toString()`

- Used to convert arrays into string format

```
import java.util.Arrays;

String[] names = { "John", "Mary", "Bob" };

System.out.println(names); // [Ljava.lang.String;@5acf9800 (memory address)
System.out.println(Arrays.toString(names)); // [John,  Mary, Bob]
```

Adding New Elements

- You cannot change the size of the array once it's created
- If more new elements are need to be added then create a new array with larger size and copy the previous elements into it using a loop
- Manually add the new elements into remaining index of the new array

```
String[] names = { "John", "Mary", "Bob" };

// Create a new array with a larger size
String[] newNames = new String[5];

// Copy the elements from the old array to the new array
for (int i = 0; i < names.length; i++) {
  newNames[i] = names[i];
}

// Update the value at index 3 and index 4
newNames[3] = "David";
newNames[4] = "Eve";
```

Reference Trap

- Variables don't store arrays, they store the reference that points to the created array in memory
- We can have more variables that store a reference which points to the same array, for example -

```
int[] nums1 = {1, 2, 3};
int[] nums2 = nums1;
```

This is a bad practice because it allows us to manipulate `nums1` through `nums2`.

```
nums2[1] = 5;
```

It will change the value of the second element for both the nums array even though we only did it for one of them. This happens since both of them are pointing to the same array in memory.

Hence, **do not set array variables equal to one another**.

> The state of a variable should not change because you updated another.

What to do when we want 2 variables to have the same array?

Create a new array for the second one and copy every value from the first into the new array using a for loop.

Note: *This can also be done using the `Array.copyOf(arrayToCopyFrom, lengthToCopy)` method.*

Otherwise, if we set array variables equal to one another, they'll point to the same array in memory and mutating one would mutate both (PITFALL).

2-D Arrays

- 2D array is an array that contains arrays
- Used when data comes in a form of table or a grid
- Usually needed when we want to create multiple arrays for the same type of items

```
// Let's say we want to store marks of 3 students of 3 subjects.
int[] student1 = { 90, 80, 70 };
int[] student2 = { 80, 70, 60 };
int[] student3 = { 70, 60, 50 };

// Store marks in 2D array.
int[][] marks = {
  { 90, 80, 70 },
  { 80, 70, 60 },
  { 70, 60, 50 }
};

// Accessing elements using row and column numbers.
System.out.println(Arrays.toString(marks[0])); // [90, 80, 70]
System.out.println(marks[0][1]); // 80

// Updating 2D Array
marks[0][1] = 85;
System.out.println(marks[0][1]); // 85
```

Looping 2-D Arrays

- Nested loops are used to loop over 2D arrays

Although we can use a normal loop for every row and get elements row by row but that isn't good if we have many rows in our 2D arrays

```
int[][] marks = {
  { 90, 80, 70 },
  { 80, 70, 60 },
  { 70, 60, 50 }
};

// Loop to print every element from the row by row
System.out.print("First row: ");
for (int i = 0; i < marks[0].length; i++) {
  System.out.print(marks[0][i] + " "); // [0][0] [0][1] [0][2]
}

System.out.print("\nSecond row: ");
for (int i = 0; i < marks[1].length; i++) {
  System.out.print(marks[1][i] + " "); // [1][0] [1][1] [1][2]
}

System.out.print("\nThird row: ");
for (int i = 0; i < marks[2].length; i++) {
  System.out.print(marks[2][i] + " "); // [2][0] [2][1] [2][2]
}
```

- It can be observed that this method isn't ideal if we have 100 rows in our `marks` 2D array
- The loops for every row are identical and hence, we can use nested loops
- One **outer loop (i)** that runs through every row and one **inner loop (j)** that runs through every element of one particular row

```
int[][] marks = {
  { 90, 80, 70 },
  { 80, 70, 60 },
  { 70, 60, 50 }
};

System.out.println("\n\nAll elements: ");

for (int i = 0; i < marks.length; i++) {
  for (int j = 0; j < marks[i].length; j++) {
    System.out.print(marks[i][j] + " ");
  }
  System.out.print("\n");
}
```

Output -

```
# Outer loop iteration: i = 0,
# Inner loop iterations: j = 0, 1, 2
marks[0][0] + " " + marks[0][1] + " " + marks[0][2] # 90 80 70

# Outer loop iteration: i = 1,
# Inner loop iterations: j = 0, 1, 2
marks[1][0] + " " + marks[1][1] + " " + marks[1][2] # 80 70 60

# Outer loop iteration: i = 2,
# Inner loop iterations: j = 0, 1, 2
marks[2][0] + " " + marks[2][1] + " " + marks[2][2] # 70 60 50
```

Small Tip on Arrays

- Return an array directly without first storing it into a variable in the following way -

```
return new Type[] { element1, element2 };
```

Objects

- Make code easy to read and understand
- Help us divide our code in modules
- Make it easy to organize code

How to identify objects?

- Something that contains fields, for eg, a car object will have fields such as *make*, *price*, *color*, etc.
- Something that can perform tasks, for eg, a car object will perform *the task of driving*

Object Oriented Programming

- Organizing your code around objects to write high-quality code.

Class

- Class is a blueprint from which objects are created
- Defines all the properties or fields the objects need to have

```
// Car.java

public class Car {
  String make;
  double price;
  int year;
  String color;
}
```

Creating Objects from Class

Now from the **Car class** we can create many **Car objects**

```
Car toyota = new Car(); // variable `toyota` stores the reference to the created object in memory
```

Once an object is created, we can modify it's fields

```
toyota.make = "Toyota"
toyota.price = 10000
toyota.year = 2020
toyota.color = "Green"
```

Unless we provide some field values, they are considered as null.

Constructor

- First thing that runs when you create an object
- Allows us create and update object properties in a single line
- The fields should be updated inside the constructor only

Creating a Constructor

- Specify a lever of access (public)
- The name of the class
- Parameters or the fields that constructor is going to update

```
public Car(String make, double price, int year, String color) {
  // properties
}
```

- When calling a constructor we need to pass the values for the fields/constructor parameters

```
Car nissan = new Car("Nissan", 10000, 2020, "Green");
Car toyota = new Car("Toyota", 20000, 2021, "Red");
```

`this` keyword

- Refers to the current object
- Used to distinguish between parameters and the fields that are updated
- `this` can be used to update the fields using the passed values for Constructor parameters

```
public Car(String make, double price, int year, String color) {
  // updating fields for that object for which `this` is used
  this.make = make;
  this.price = price;
  this.year = year;
  this.color = color;
}
```

When the constructor is done updating the field values using the values passed for the constructor parameters, our object is fully initialized (created) and the variable used stores the reference to this created object.

Getters

- A public method used to get the value of some field of a class
- It's return type is always the type of the field whose value it returns
- The method name always starts with **get** followed by field name

```
public String getMake() {
  return make;
}
```

Why Getters?

- If we don't make the class fields **private**, they can be accidentally updated later

```
Car nissan = new Car("Nissan", 10000, 2020, "Green");

// accidentally update
nissan.make = "Toyota";
```

- It's considered a bad practice and hence we should always make the class fields **private** to that class only

```
public class Car {
  // private fields
  private String make;
  private double price;
  private int year;

  public Car(String make, double price, int year,) {
    this.make = make;
    this.price = price;
    this.year = year;
  }
}
```

- This makes the fields private but it means we can't access them even while printing

This is where **Getters** come in. They are methods which are used to get the value of certain fields.

```
public String getMake() {
  return make;
  // We don't need `this` keyword since there is no conflicting variable of the same name unlike we had in the constructor function
}
```

- Always use getters to access object fields

Setters

- Public methods used to update the private fields of a class
- Takes a parameter whose type depends on the field it's going to update
- Name always starts with **set** followed by the field name it updates

```
public void setMake(String make) {
  this.make = make;
}
```

Why Setters?

- Since fields are made private using the **private** keyword and hence they can not be directly updated using the objects

The following code won't work since the fields are not visible to *toyota* object -

```
Car toyota = new Car("Toyota", 20000, 2021, "Red");

toyota.make = "Toyota"; // won't work
toyota.price = 20000; // won't work
```

That's where **Setters** come in. These methods are used to update the value of certain fields.

```
public void setMake(String make) {
  this.make = make;
  // We need `this` keyword since there is a conflicting variable of the same
}
```

Now even though we have private fields in our class, we have public Getters to get the field values and public setters to update the field values.

> There's a nifty VSCode Extension called **Java Code Generators** that can generate getters and setters for any class with just one command.

Copy Constructor

- Copy Constructor is used to create an exact copy of an object with the same values of an existing object.

Let's say we need to create an object which has to be the same as another created object. We can't make the 2 objects equal to each other otherwise we'd run into reference trap. For example,

```
Car nissan = new Car("Nissan", 10000, 2020, "Green");
Car nissan2 = nissan; // □ wrong practice

// Since both `nissan` and `nissan2` point to the same object in the memory, updating any one would update both of them
nissan2.setColor("Yellow");
nissan.getColor(); // Yellow
nissan2.getColor(); // Yellow

nissan.setColor("Orange");
nissan.getColor(); // Orange
nissan2.getColor(); // Orange
```

When we're in a situation like this, a **copy constructor** can be used.

- Used to create an object with the properties of another object

```
// Constructor
public Car(String make, double price, int year, String color) {
  this.make = make;
  this.price = price;
  this.year = year;
  this.color = color;
}

// Copy Constructor
public Car(Car source) {
  this.make = source.make;
  this.price = source.price;
  this.year = source.year;
  this.color = source.color;
}
```

- While creating the new object we should use the copy constructor instead so that both objects don't share the same reference

```
Car nissan = new Car("Nissan", 10000, 2020, "Green"); // using Constructor
Car nissan2 = new Car(nissan); // using Copy Constructor
```

- Now the variable `nissan2` will store a unique reference

```
nissan2.setColor("Yellow");
nissan.getColor(); // Green
nissan2.getColor(); // Yellow

nissan.setColor("Orange");
nissan.getColor(); // Orange
nissan2.getColor(); // Yellow
```

Mutable Objects

- Objects which can be mutated using setters after they are created
- Dealing with these objects, avoid setting them equal to one another

When to use Copy Constructors?

There are 2 good reasons for using a copy constructor instead of the constructor passing all parameters:

1. When you have a complex object with many attributes it is much simpler to use the copy constructor instead of creating the same/similar object again and again with so many attributes and values.
2. If you need to add a few new attributes to your objects along with older ones, you just change the copy constructor to take these new attribute into account along with older ones

toString Method

If we print an object directly, we get the class it's created from along with the reference.

```
Car nissan = new Car("Nissan", 10000, 2020, "Green", spareParts);
System.out.println(nissan); // Car@36baf30c
```

This happens because whenever we print the class, behind the scenes a `toString()` method is executed and it returns a string representation of the object.

```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

We can override this `toString()` method in our class to print a more meaningful result as per our needs.

```
public String toString() {
  return "Make: " + this.make + "\n"
    + "Price: " + this.price + "\n"
    + "Year: " + this.year + "\n"
    + "Color: " + this.color + "\n"
    + "Parts: " + Arrays.toString(this.parts) + "\n";
}
```

Now when we try to print an object -

```
Car nissan = new Car("Nissan", 10000, 2020, "Green", spareParts);
System.out.println(nissan);
// Make: Nissan Altima
// Price: 11000.0
// Year: 2020
// Color: Green
// Parts: [Tires, Keys]
```

Arrays are Mutable Objects

Arrays are mutable objects that means they can be updated even after they are created. Therefore, it's very important to use them carefully. For example,

```
private String make;
private double price;
private String[] parts;

// Constructor
public Car(String make, double price, String[] parts) {
  this.make = make;
  this.price = price;
  this.parts = parts;
}
```

This would be considered a bad practice since we are making one array variable equal to the other

```
this.parts = parts; // we are just storing the reference of the `parts` array to `this.parts`
```

In this case, if we update any one of the 2 values later, both of them would be updated since they both point to the same reference in memory.

Let's say the parts array that was passed to the constructor was initially -

```
String[] spareParts = new String[] {"Tires", "Keys"};
```

and then we changed an element -

```
spareParts[0] = "Filter";
```

Since, spareParts, **parts** and **this.parts** all store the reference to the same array, all of them would be updated which isn't ideal at all.

Fixing the Constructor

It can be fixed by using a copy of the passed array as it creates a new reference (object) in the memory

```
this.parts = Arrays.copyOf(parts, parts.length); // copy full length of the array
```

Now even if we mutate the array passed to the constructor (spareParts), the car objects won't be mutated since they all store a unique reference for the parts field

Fixing the Copy Constructor

If we don't fix the copy constructor, any object created with it will share the reference (for the *parts* field) of the object that is passed as the **source**.

Then if we update the source object's or the newly created object's *parts* field, both of them would be updated since they both point to the same object in memory.

```
public Car(Car source) {
  this.make = source.make;
  this.price = source.price;
  this.year = source.year;
  this.color = source.color;
  this.parts = Arrays.copyOf(source.parts, source.parts.length); // FIX: copyOf creates a new reference in memory
}
```

Fixing the Getter

If we return the **parts** array directly from the getter than whatever variable is used will store the reference of it and hence will be able to mutate the **parts** array directly.

For example, if the getter is -

```
public String[] getParts() {
  return this.parts;
}
```

And it is used as -

```
String[] carParts = nissan.getParts();
carParts[0] = "Filters"; // will mutate `carParts` as well as `nissan.parts`
```

Here, **carParts** and **Nissan.parts** point to the same reference. Hence, mutating one would mutate both of them which is BAD PRACTICE.

This **can be fixed by returning a copy of the array from the getter** function so that a new reference would be returned.

```
public String[] getParts() {
  return Arrays.copyOf(this.parts, this.parts.length);
}
```

Now whatever variable is used to store the return value of getter function, it will hold a new reference of the array.

Fixing the Setter

For the setters, the situation is the same as we had for the constructor. We can't directly set constructor's **parts** array equal to the one that's passed, otherwise if the passed array is later changed, it will also mutate all the instances of the constructor since the passed array and the instances will both be sharing the reference to the same array in memory.

Bug -

```
public void setParts(String[] parts) {
  this.parts = parts;
}
```

Fix -

```
public void setParts(String[] parts) {
  this.parts = Arrays.copyOf(parts, parts.length);
}
```

This will make the parts field store a unique reference of the passed array every time the setter function is used.

Immutable Objects

Data types in java can be categorized into 3 types -

1. Primitive
2. Immutable Objects
3. Mutable Objects

Primitive Data Types

- Most basic data type in Java
- Variables store a single value

```
int apples = 6; // 4 bytes of memory
long population = 6000000000000L; // 8 bytes of memory
double price = 5.99; // 8 bytes of memory
char = 'A'; // 2 bytes of memory
boolean bool = true; // 1 bit of memory
```

Immutable Objects

- For every primitive data type, there is an immutable object type
- We can create new objects of any primitive type using that type's class
- Variables used will only store reference to that object
- Works in the following way -

```
Integer apples = new Integer(5);
```

- Above method is now depreciated so we can do it directly

```
Integer apples = 5; // 16 bytes
Long population = 6000000000000L; // 24 bytes
Double price = 5.99; // 24 bytes of memory
Char = 'A'; // 16 bytes
Boolean bool = true; // 16 bytes
```

Immutable Objects vs Primitive

1. Immutable objects take more memory than primitive types
    - Include object metadata along with the field value itself
2. Immutable objects can be null, primitive can't be null
    - `null` is a reference that points to nothing when there is no object

    ```
    Integer apples = null;
    apples = 5;
    ```

    - Primitive types don't play with reference so they can't ever hold `null` since they won't ever eventually point to any object
3. Immutable objects can call methods, primitive objects can not

Hence, only use immutable objects if you have to otherwise always prefer primitive.

Immutable Objects vs Mutable Objects

1. Immutable Objects are **safer** since they cannot be modified after creation
    - State of an immutable object can't be changed
    - References can be shared safely across your application
    - You cannot update an immutable object. You can only set the variable equal to a brand new immutable object.
    - Once an immutable object is updated, a brand new reference is created

    ```
    Integer apples = 5; // apples = Integer@10
    Integer apples2 = apples; // apples2 = Integer@10
    apples2 = 10; // apples2 = Integer@18
    ```

2. Mutable Objects are **less safe** since they can be modified
    - Avoid setting 2 variables equal to one another
    - Sharing references can lead to unintended side effects

```
City city = new City("Paris"); // city = Object@10
City city2 = city; // city2 = Integer@10
city2.setName("Shimla") = 10; // city2 = Integer@10, city = Integer@10
// `City` also becomes Shimla
```

String is an Immutable Object

```
String text = "hi";
String text2 = new String("Hello");
```

Here we are basically setting our variable equal to a new object of the String class.

List Collections

- Used for storing collections of data
- Normal arrays have a fixed size and when we want to add more elements, we'd need to create a new array of bigger size
- Lists like **ArrayList** and **LinkedList** can be used to solve this problem

ArrayList & LinkedList

- Do not have a fixed size
- Store data differently
- **ArrayList** is generally preferred over **LinkedList**
- Both are of type `List`

Creating Lists

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

List list = new ArrayList();
List list2 = new LinkedList();
```

> When two objects can share the same type, we call this **polymorphism**.

- Type of element a list can store is called **generic**
- Can be overridden by using pointy brackets (`<>`)

```
List<String> list = new ArrayList<>();
```

This tells Java that we want our list to store elements of type String.

Adding Elements to Lists

- All list collections have an `add()` method to add elements

```
List<String> cities = new ArrayList<>();
cities.add("Paris");
```

Removing Elements from Lists

- All list collections have a `remove()` method to remove elements

```
List<String> cities = new ArrayList<>();
cities.add("Paris");
cities.add("Florence");

cities.remove("Paris"); // removes `Paris`
```

More List Methods

- `get()` - Used to get an element by index

```
cities.get(5);
```

- `set()` - Used to set/replace an element using index and new element

```
cities.set(3, "Jaipur");
```

- `size()` - To get the size of the list

```
cities.size();
```

ArrayList

- Every time we add an element to an ArrayList Java checks whether list **size** is equal to the **capacity**
- Initially both of them are 0, so when we add an element Java sets the underlying field equal to a new array that has the capacity to store 10 elements
- Now if we have added 1 element to the list, **capacity is 10** but the **size is 1**
- When list **size** is finally equal to **capacity** and we add another element, the array list is smart enough to dynamically resize by setting its field equal to a new array of a larger capacity

Thus, when we are using ArrayLists, we can add as many elements as we want since it's smart enough to resize.

```
main() {
  List<String> cities = new ArrayList<>();
}
```

LinkedList

- Just like ArrayList, they don't have a fixed size
- Stores data very differently than ArrayList

```
import java.util.LinkedList;
import java.util.List;

main() {
  List<String> cities = new LinkedList<>();
}
```

- Adding an element to a LinkedList creates a new node that stores the element
- When first node is created, it contains a reference that uniquely identifies it
- When we add more elements to the list, more new nodes are created which not only store the added element but also store the reference to their previous node
- In turn, the previous node stores a reference to the new node

The more elements we add the more new nodes are created such that each nodes store the added element and the reference to their previous and next node.

Why retrieving elements from a LinkedList is slow?

Retrieving data from a LinkedList is very slow compared to ArrayLists

- ArrayLists are backed by normal arrays so if we want to access something, it can be grabbed via that index
- LinkedLists don't have any index, they only have a chain of nodes
- So if we want to access any element, the list needs to traverse from the first or the last node (based on which is closer) all the way to the required node one by one.

Hence, it is comparatively slow.

ArrayList vs LinkedList

- ArrayList is backed by a regular array that increases its capacity based on array size
- LinkedList comprises of node where each node is linked to the prev and the next node

Which one is more efficient?

- **ArrayList**: retrieving elements
- **No Difference**: Performing operations from either end of the list
- **LinkedList**: add/remove elements from the middle

While inserting an element in the middle, the ArrayList has to create a new array and copy over the elements of the prev array while inserting the new element inside. It can't just insert a new element in the middle.

The same thing happens while removing an element from the middle.

Here's a small example of different operations and their recorded time -

```
# FETCHING DATA
LinkedList: 43ms
ArrayList: 0ms

# FIRST / LAST INDEX
LinkedList: 0ms
ArrayList: 0ms

# INSERTION FROM MIDDLE
LinkedList: 0ms
ArrayList: 12ms

# INSERTION
LinkedList: 0ms
ArrayList: 0ms

# REMOVAL
LinkedList: 0ms
ArrayList: 14ms

# UPDATES
LinkedList: 56ms
ArrayList: 0ms
```

Default equals()

- Only checks for references of objects and whether they are equal or not
- Uses `==` operator to compare references

```
City paris = new City("Paris", 2161000);
City copy = paris;
City differentCopy = new City(paris);

System.out.println(paris.equals(copy)); // `true` since both share references
System.out.println(paris.equals(differentCopy)); // `false` since copy constructor is used hence a new copy and reference of the `pa
```

- So even if all the values are equal, it'll return `false` if the references are different

The issue with ArrayList

- Every time we add an object to the list, a new reference is created

```
ArrayList<City> cities = new ArrayList<>();
cities.add(new City("Paris", 2161000)); // new reference
cities.add(new City("Florence", 382258));
cities.add(new City("Venice", 261905));
```

- Let's say we check if a certain object is present is in the list or not

```
 City paris = new City("Paris", 2161000); // new reference
System.out.println(cities.contains(paris)); // false
```

Now even though the list has an object with the same fields we asked, it still returns false since the references of both the objects are different. This happens because behind the scenes the `contains()` method loops over the list to check uses **default equals** which only checks for references.

Hence, this isn't ideal since our list does contain the asked object.

Overriding equals()

- Default equals method can be overridden in it's class

```
@Override
public boolean equals(Object o) {
  if (o == this)
      return true;
  if (!(o instanceof City)) {
      return false;
  }
  City city = (City) o;
  return Objects.equals(name, city.name) && population == city.population;
}

// > Equality Contract: When you override equals, you must override hashcode.
@Override
public int hashCode() {
  return Objects.hash(name, population);
}
```

- DON"T change signature (parameter name) of the equals method otherwise Java won't recognize you're overriding it
- In the parameters, `Object` is the first argument that can be an object of any type (including Lists) and `o` is the second argument which is being compared with the first one

Working

- First, we check if the references of the objects are same

```
if (o == this)
  return true;
```

- Now, if the references of the object are not equal, we check if the second object is not an instance of the same class

```
if (!(o instanceof City)) {
  return false;
}
```

- If it is instance of the same class, then we typecast it first and then compare all the field values of both objects

```
City city = (City) o;
return Objects.equals(name, city.name) && population == city.population;
```

So now if we test the following code -

```
ArrayList<City> cities = new ArrayList<>();
cities.add(new City("Paris", 2161000)); // new reference
cities.add(new City("Florence", 382258));
cities.add(new City("Venice", 261905));

City paris = new City("Paris", 2161000); // new reference
System.out.println(cities.contains(paris));
```

- `contains()` method will call overridden `equals` method
- First condition will be false since references aren't equal
- Second condition will also pass since `paris` object is an instance of `City` class
- Then we compare the field values `paris` object with the city object at each iteration and return true when all the field values match
- So we finally check if an Object (List in our case) contains another object even if the references are unique

Exception Handling

There are two types of exceptions -

1. **Compile Time**: predictable failure checked before execution
2. **Runtime**: occurs at runtime and isn't checked by compiler

Compile-Time Exceptions

- Failures that we can predict before executing our code
- Examples are -
    - `FileNotFoundException`
    - `NetworkException`
    - `SQLException`
    - `IOException`

Let's say we want to load an external file in our java code

The basic approach would be the following -

```
import java.io.FileInputStream;

public class Example1 {
  public static void main(String[] args) {
    FileInputStream fis = new FileInputStream("greetings.txt");
  }
}
```

This code will not run since it's possible that the **greetings.txt** file may randomly be deleted at any time and hence we won't be able to load it.

Hence we need to use the **try and catch block** to run this code.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class Example1 {
  public static void main(String[] args) {
    // try to run the code
    try {
      FileInputStream fis = new FileInputStream("greetings.txt");
    }
    // incase we get a FileNotFound error, get the message from the error
    catch (FileNotFoundException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

- So instead of crashing, it allows us to fail on our own terms
- Allows us to handle the errors based on how we want

`Finally` Block

- runs no matter what
- good place to clean up code

```
main() {
  try {
    FileInputStream fis = new FileInputStream("greetings.txt");
  }
  catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
  }
  finally {
    System.out.println("Process complete");
  }
}
```

The above **finally** block runs no matter if we run the **try** or the **catch** block

Runtime Exception

- Exceptions not checked by the compiler
- Examples are -
    - `ArrayIndexOutOfBoundsException`
    - `NullPointerException`
    - `InputMismatchException`

    - `IllegalArgumentException`
- These aren't always result of programming errors
- It is developer's responsibility to expect these and write error-prone code

Examples

ArrayIndexOutOfBoundsException

```
int[] grades = new int[] { 96, 65, 56, 86 };
System.out.println(grades[4]); // element 4 isn't there
```

InputMismatchException

```
String[] names = new String[7];
names[0] = "John";
names[1] = "Jim";
names[2] = "Joe";

for (String name : names) {
  System.out.println(name.toUpperCase()); // calling methods from null after 3 iterations
}
```

InputMismatchException

```
Scanner scanner = new Scanner(System.in);
System.out.println("Please enter a random integer");
scanner.nextInt(); // mismatch when user enters a wrong type like string
scanner.close();
```

Argument Validation

- Ensures that methods only receive correct arguments

Quality Control POV

- Throw `IllegalArgumentException` if a method receives wrong arguments

```
public void setAge(int age) {
  if(age < 0) {
    throw new IllegalArgumentException("Age cannot be negative");
  }
  this.age = age;
}

public void setUsername(String username) {
  if(username == null || username.isBlank()) {
    throw new IllegalArgumentException("Invalid username");
  }
  this.age = age;
}
```

- Ensure our methods only run if we pass the right arguments, otherwise we crash the program

Handling User Input POV

- Write code that prevents exceptions from being thrown

The following code will throw an error since the `username` is initially null

```
user.setUsername(user.getUsername());
```

A correct way could be the following but it'll still crash if user enters a blank value

```
user.setUsername(scanner.nextLine());
```

Hence, the best way to predict and handle exceptions would be -

```
String username = scanner.nextLine();

if (username.isBlank()) {
  System.out.println("Sorry, that is an invalid username");
} else {
  user.setUsername(username);
}
```

Similarly, we can validate for **age** too -

```
String age = scanner.nextInt();

if (age < 0) {
  System.out.println("Sorry, that is an invalid age");
} else {
  user.setUsername(age);
}
```

Map Collections

HashMap: stores unordered collection of key-value pairs

TreeMap: stores ordered collection of key-value pairs

- Both share the same type `Map`

> Both objects sharing same type -> **Polymorphism**

- Maps can share any pair of objects like "Strings and Integers", "Integers and Strings", "Strings and Decimals"
- First object in every entry is called **key** and it maps to another object called a **value**

Creating Maps

```
import java.util.Map;
import java.util.HashMap;
import java.util.TreeMap;

public class HashMapInventory {
  public static void main(String[] args) {
    // HashMap
    Map<String, Double> inventory = new HashMap<>();

    // TreeMap
    Map<Integer, Integer> tree = new TreeMap<>();
  }
}
```

Adding Entries

Map collections have a method called `put()` which accepts key-value pairs

```
map.put("Bananas", 3.99);
```

Getting Entries

Map collections have a method called `get()` which accepts key and returns its mapped value

```
map.get("Bananas"); // 3.99
```

Checking Entries

Maps have a `containsKey()` method which checks whether a key exists in collection or not by returning a boolean value

```
boolean result = inventory.containsKey("Apples");
```

HashMap

Stores unordered collection of entries

Let's store price of different items in a Hashmap where each item will be of string type and its price will be of double type -

```
 // Create a HashMap
Map<String, Double> inventory = new HashMap<>();
```

How data gets stored?

```
 // Insert Values
inventory.put("Bananas", 3.99);
inventory.put("Papaya", 5.99);
inventory.put("Kale", 4.99);
inventory.put("Apples", 1.99);
```

1. The moment we add the first entry, an array that stores 16 elements gets created.

```
# New array created
```

2. A function processes the key and returns an integer known as a **hash**.

```
# Key is processed and a hash value of `14565564` is returned
```

3. Then a bitwise operation on the hash determines the index where the key-value pair will be stored.

```
# Bitwise operation converts `14565564` to an index `3`
```

4. The element at that index stores a reference to a node that stores the key-value entry.

```
# A new element is created at index `3` which stores a reference to a node with key-value `bananas: 3.99`
```

5. The process repeats every time we add a value to the HashMap
6. If we get an index where an element is already stored, a collision occurs and in this case, the prev node at that index now stores a reference to the new node (imagine LinkedList)

Hence, in HashMaps we have an array where each element of the array refers to a linked list of nodes.

How data is retrieved?

```
 // Insert Values
inventory.get("Bananas"); // 3.99
inventory.get("Papaya"); // 5.99
```

1. We provide a key in `get()` method
2. It gets hashed into a numerical value
3. A bitwise operation determines the index in the array using that numerical value
4. We check the linked list at that index and find the node which matches our key
5. Once we find a key, we get its corresponding value

When there are no collisions, **lookup operations** on HashMaps are very fast.

TreeMap

Stores ordered collection of entries

Let's create a multiplication table using TreeMap for `5` where key is the multiplier (integer) and value is the multiple (integer) -

```
// Create a TreeMap
Map<Integer, Integer> tree = new TreeMap<>();
```

Adding Entries

```
int n = 5;

tree.put(1, n * 1); // 1: 5
tree.put(2, n * 2); // 2: 10
tree.put(3, n * 3); // 3: 15
tree.put(4, n * 4); // 4: 20
tree.put(5, n * 5); // 5: 25
tree.put(6, n * 6); // 5: 30
tree.put(7, n * 7); // 5: 35
```

- When we add entries to a TreeMap, it stores them in a tree-like data structure
- It uses *the red-black tree algorithm* to continuously sort entries as they are being added
- It basically stores the entries in ascending order of the keys
- As a result, every node in the left sub-tree is going to be smaller than its root sub-node and every node in the right sub-tree is going to be higher than the root sub-node

```
/**
 *         20
 *       /   \
 *     10     30
 *   /   \    / \
 *  5    15  25  35
 */
```

If we go from left to right, we see tree maps are sorted in ascending order.

Accessing Entries

```
System.out.println(tree.get(4)); // 20
System.out.println(tree.get(7)); // 35
```

- Whenever we get a key we go to the main root node and check whether given key is lower or higher
- If it's lower, we go to the left child nodes else if it's higher we go right
- We keep repeating the step until we find the key we're looking for
- The deeper our node is, the more steps and time it will take to retrieve it

Hence, we see that **TreeMaps** sort entries in ascending key order and the time taken to retrieve data depends on the tree depth.

> Use **HashMap** if order doesn't matter and **TreeMap** if entries need to be sorted 0

HashMap Equality

HashMaps by default rely on default equals and hashcode methods which can be problematic in some cases as they use references for operations. Hence, we need to update these default methods.

`hashcode()` method

- By default returns a hashcode value based on the reference

```
Contact contact = new Contact("Alice", 30);
Contact contactCopy = contact;
Contact contactCopy2 = new Contact(contact);

contact.hashCode(); // 918221580
contactCopy.hashCode(); // 918221580
contactCopy2.hashCode(); // 456466518
```

- For the first 2 cases, both hash codes are same since both the objects point to the same reference in memory

Default `equals()` method

- By default the `equals` method compares references

```
Contact contact = new Contact("Alice", 30);
Contact contactCopy = contact;
Contact contactCopy2 = new Contact(contact);

contact.equals(contactCopy) // true
contact.equals(contactCopy2) // false
```

- Method returns **true** for the first one and **false** for the second since it compares references of both objects

Problem with this approach

Let's say we create a HashMap and insert a bunch of objects in it

```
Map<Contact, String> peopleMap = new HashMap<>();

peopleMap.put(new Contact("Alice", 30), "1806 Farm Meadow Drive");
peopleMap.put(new Contact("Bob", 35), "4046 Weekley Street");
peopleMap.put(new Contact("Charles", 36), "1110 Cerullo Road");
```

Now let's say we want to retrieve an object from the map

```
peopleMap.get(new Contact("Alice", 30)) // returns `null`
```

The above line returns `null` since whenever we create a new object of the `Contact` class, a new reference is created, so when we try to retrieve it with the new reference it doesn't match with the one store in the Map.

Fix for the above issue

This above issue is not at all ideal. Hence, we need to override the default `equals` and `hashcode` methods.

```
@Override
public boolean equals(Object o) {
  if (o == this)
    return true;
  if (!(o instanceof Contact)) {
    return false;
  }
  Contact contact = (Contact) o;
  return Objects.equals(name, contact.name) && age == contact.age;
}

@Override
public int hashCode() {
  return Objects.hash(name, age);
}
```

Comparing two objects by references now

```
Contact contact = new Contact("Alice", 30);  // Reference -> R1
Contact contactCopy = contact;               // Reference -> R1
Contact contactCopy2 = new Contact(contact); // Reference -> R2

// HashCode()
contact.hashCode();       // 918221580
contactCopy.hashCode();   // 918221580
contactCopy2.hashCode();  // 918221580

// Equals
contact.equals(contactCopy) // true
contact.equals(contactCopy) // true
```

This happens since the overridden methods now don't compare based on the references, instead, they compare the objects based on their field types.

Trying with HashMap

```
Map<Contact, String> peopleMap = new HashMap<>();

peopleMap.put(new Contact("Alice", 30), "1806 Farm Meadow Drive");
peopleMap.put(new Contact("Bob", 35), "4046 Weekley Street");
peopleMap.put(new Contact("Charles", 36), "1110 Cerullo Road");
```

Now let's say we want to retrieve an object from the map

```
peopleMap.get(new Contact("Alice", 30)) // returns `1806 Farm Meadow Drive`
```

The above line returns the expected value since even though a new reference is created, we have overridden the behind the scene's hash method to check for the values based on fields entered.

Inheritance

This is one of the most important aspects of OOPs.

- Allows parent class to serve as a foundation for several child classes
- Every child class inherits fields and methods from its parent

It allows us to reuse code instead of writing it over and over again.

Example

If 2 classes (Let's say `Shirt` and `Pants`) have a few same fields and methods (size, prices, etc.), instead of writing them in both the classes, we can create a parent class (`Product`) which can have all these fields and methods so that the child classes can inherit from it.

Product.java

```
public class Product {
  private double price;
  private String color;

  // # Getters
  public double getPrice() {
    return this.price;
  }

  public String getColor() {
    return this.color;
  }

  // # Setters
  public void setPrice(double price) {
    this.price = price;
  }

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

Use the `extend` keyword to make the child class extend to the parent class

Shirt.java

```
public class Shirt extends Product {
  private Size size;

  public enum Size {
    SMALL, MEDIUM, LARGE
  }

  public Size getSize() {
    return this.size;
  }
  public void setSize(Size size) {
    this.size = size;
  }
}
```

Pants.java

```
public class Pants extends Product {
  private int length;

  public int getLength() {
    return this.length;
  }
  public void setLength(int length) {
    this.length = length;
  }
}
```

Now `Shirt` and `Pants` will inherit all the fields and methods from the `Product` class and they can be used the same as before

Main.java

```
main() {
  Shirt shirt = new Shirt();
  // methods inherited from Shirt (child) class
  shirt.setSize(Size.SMALL);
  // methods inherited from Product (parent) class
  shirt.setColor("Black");

  Pants pants = new Pants();
  // methods inherited from Pants (child) class
  pants.setWaist(32);
  pants.setLength(32);
  // methods inherited from Product (parent) class
  pants.setColor("Navy Blue");
  pants.setBrand("Nike");
}
```

Polymorphism

- Ability of an object to take on multiple forms
- Child objects can take the form of their parent object

Example

The child class `Shirt` can also be considered a `Product`

```
// Shirt shirt = new Shirt();
Product shirt = new Shirt();
```

However, it is recommended to always use the most specific class instead of the most generic class since the specific child class (`Shirt` in our case) can also have methods of its own and they can't be called by the generic parent class.

```java
Product shirt = new Shirt();
shirt.setSize(Size.SMALL); // ▯ error

Shirt shirt = new Shirt();
shirt.setSize(Size.SMALL); // ▯ works
```

Using specific class ensures that the class can access every method from the parent class plus the additional ones from the child class.

Why do we need Polymorphism?

- Flexible and reusable code

Now we don't have to create separate methods for both `Shirt` and `Pants` for a similar task. For example, instead of writing methods like this -

Main.java

```java
public static void pantStore(Pants pants) {
  System.out.println("Thank your for purchasing " + pants.getBrand() + " pants!");
}

public static void shirtStore(Shirt shirt) {
  System.out.println("Thank your for purchasing " + shirt.getBrand() + " shirt!");
}
```

We can use something like this -

Main.java

```java
public static void productStore(Product product) {
  System.out.println("Thank your for purchasing " + product.getBrand() + " " + product.getClass().getSimpleName() + "!");
}
```

Instead of using the child class-specific getters, we can use the parent class getters which are inherited by all child classes.

`super` keyword

- Refers to the parent class of a class
- `super()` - Invokes the parent constructor to update the inherited fields
- `super.member` - Used to access member (fields/methods) of a parent class from any child class

Here, the parent class expects 3 arguments for the fields that will be inherited by it's child classes

Product.java

```java
private double price;
private String color;
private String brand;

// # Constructor
public Product(double price, String color, String brand) {
  this.price = price;
  this.color = color;
  this.brand = brand;
}
```

But both the child classes need to have all arguments including the child-specific fields and the parent class fields

Shirt.java

```
private Size size;

public enum Size {
  SMALL, MEDIUM, LARGE
}

// Constructor
public Shirt(double price, String color, String brand, Size size) {
  this.size = size;
}
```

Pants.java

```
private int waist;
private int length;

// Constructor
public Pants(double price, String color, String brand, int waist, int length) {
  this.waist = waist;
  this.length = length;
}
```

Now when we get all these arguments while creating an object from the class, we need to somehow pass the parent fields to the parent class since the child class only use the child class specific arguments. This is where `super` keyword is used.

```
super(price, color, brand);
```

We pass all the required arguments while creating the objects

Main.java

```
Shirt shirt = new Shirt(100.0, "Black", "Adidas", Size.SMALL);
Pants pants = new Pants(120.0, "Navy Blue", "Nike", 32, 56);
```

The child class gets the arguments and passes them to the parent class

Shirt.java

```
public Shirt(double price, String color, String brand, Size size) {
  super(price, color, brand);
  this.size = size;
}
```

Pants.java

```
public Pants(double price, String color, String brand, int waist, int length) {
  super(price, color, brand);
  this.waist = waist;
  this.length = length;
}
```

Now the 3 arguments are passed to the `super` constructor which means the constructor of the current class's parent class which is `Product` class in our case.

These 3 fields are updated when the parent constructor is called and the rest of the child-specific fields are updated by the respective child class constructors.

> Parent class is also called `super` class

Overriding a Method

- Parent class methods can be overridden in the child class
- Newly overridden method replaces the previous parent one

Example

We had a fold method in Parent class `Product` which folds the clothes.

Product.java

```
public void fold() {
  System.out.println("Folding my " + this.brand + " " + this.color + " " + this.getClass().getSimpleName() + "...");
}
```

But both the child classes `Shirt` and `Pants` have different ways of being folded so we need to override the parent method.

Shirt.java

```
@Override
public void fold() {
  System.out.println("Lay shirt on a flat surface");
  System.out.println("Fold the shirt sideways");
  System.out.println("Bring sleeves in");
  System.out.println("Fold from bottom to top");
}
```

Pants.java

```
@Override
public void fold() {
  System.out.println("Hold pants by waist");
  System.out.println("Fold pants in half");
  System.out.println("Fold pants from bottom to top");
  System.out.println("Fold pants in half");
}
```

Now whenever we call the `fold` method from `Shirt` or the `Pants` class, the overridden methods will be called.

`@Override`

- Checks to make sure you are actually overriding a parent method
- Helps reduce programming errors

Abstract

Used to create **abstract class** and **abstract methods**

Abstract Class

- A class you cannot create objects from

For example, right now the `Product` class is only acting as a parent class to give inheritance to child classes.

If we create an instance object from it -

```
Product product = new Product(19.99, "Black", "Puma");
```

We will get a `product` object we don't know whether it's a shirt or a pant. Hence, there is no need to create objects from the `Product` class and we shouldn't allow it.

We can do this by making it an **abstract** class -

```
public abstract class Product {
  private double price;
  private String color;
  private String brand;
  // more code...
}
```

Now if we try to create an object from the **abstract** `Product` parent class, we will get a compile-time error.

Abstract Method

- Can only be defined inside an **abstract class**
- Methods that must be overridden by the child classes
- Specifies what should be returned and what parameters to expect
- Doesn't have a body since its implementation depends on the child class completely

Defined in the abstract class only

Product.java

```
 // Methods
public abstract void wear();
```

Child classes are forced to override them

Shirt.java

```
 @Override
public void wear() {
  System.out.println("Wearing my " + this.getBrand() + " " + this.getColor() + " Shirt...");
}
```

Pants.java

```
 @Override
public void wear() {
  System.out.println("Wearing my " + this.getBrand() + " " + this.getColor() + " Pants...");
}
```

The overridden abstract methods can have implementation as specific as needed for that particular child class.

The `Object` Class

- Root parent of every other class in Java
- Defines methods like `equals()`, `toString()`, etc.
- Since it is the root class for all classes, they all inherit its methods
- These methods can be overridden too as we do with `toString()` to print an object in a readable and useful form.

Higher Order Functions

- Makes code a lot more
- Need to know your intent before running
- Lambda expressions are used to express that intent
- Takes in a lambda function to specify what task to perform
- Depend/Rely on different types of functional interfaces for the lambda functions

The Benefit of Higher Order Functions

1. *Conciseness*: provide abstraction over iteration and operations, resulting in more concise and maintainable code
2. *Reusability*: offer reusability with any collection, enabling the creation of a single function that can be utilized throughout the codebase
3. *Expressiveness*: simplify code and enhance its expressiveness by providing functional methods, such as map, that can easily transform data
4. *Function composition*: can be combined to create more complex operations, enabling the creation of larger, more modular code
5. *Declarative programming*: promote declarative programming, allowing developers to specify what should happen without concerning themselves with implementation details

Types of Lambda Functions

Functional Interface	Example
Consumer	`x -> {code}`
Predicate	`x -> { return boolean }`
Function	`x -> { return value }`
BiConsumer	`(x, y) -> { code }`
Comparator	`(x, y) -> { return int }`

Functional Interface	Example
BinaryOperator	`(x, y) -> { return value (same type) }`

Consumer

Receives a parameter and produces a side-effect - `x -> {code}`

`ForEach` when invoked from an array relies on a `Consumer`

```
 List<String> facts = Arrays.asList(
    "Marie Curie was the first woman to win a Nobel Prize.",
    "She received the Nobel Prize in Physics in 1903, and the Nobel Prize in Chemistry in 1911.",
    "Curie was a pioneer in the field of radioactivity, and her work led to the development of X-ray technology."
  );
                    //  consumer
facts.forEach(fact -> System.out.println(fact));
```

Comparator

Receives two parameters and returns an integer - `(x, y) -> {return int}`

`sort` relies on a `Comparator`

```
 List<Integer> integers = Arrays.asList(25, 19, 23, 45, 38, 23, 59, 12);

// Sort the list of integers in ascending order
integers.sort((right, left) -> {
  return right.compareTo(left);
});

// Sort the list of integers in descending order
integers.sort((right, left) -> left.compareTo(right));
```

BiConsumer

Receives two parameters and produces a side-effect - `(x, y) -> {code}`

`ForEach` when invoked from a Map relies on a `BiConsumer`

```
 List<Integer> integers = Arrays.asList(25, 19, 23, 45, 38, 23, 59, 12);

// Sort the list of integers in ascending order
integers.sort((right, left) -> {
  return right.compareTo(left);
});

// Sort the list of integers in descending order
integers.sort((right, left) -> left.compareTo(right));
```

Stream Operations

- Make code expressive and easier to read
- Most stream operations are higher-order functions that do most work behind the scenes
- Chaining these operations makes very complex logic into a couple of lines of code

Stream Process

- Stream pipelines cannot process entire data source at once
- Data source need to be divided into streams

```
 List<Double> prices = Arrays.asList(341.67, 209.32, 88.41, 269.99, 68.49, 499.99, 28.12, 354.38);

// break data source into a stream of elements
prices.stream();
```

- Stream pipeline process streams one element at a time

```
prices.stream()
  .filter(price -> {
    return price > 100;
  })
  .map(price -> {
    return price * 0.9;
  })
  .sorted((price1, price2) -> {
    return price1.compareTo(price2);
  })
  .map(price -> {
    return "$" + price;
  })
  .forEach(price -> {
    System.out.println(price);
  });
```

Here `forEach` is the **terminal operation** since it terminates the pipeline after producing the output.

- Java is smart enough to infer *the return type* based on the type of lambda expression

So our code can be simplified into -

```
prices.stream()
  .filter(price -> price > 100)
  .map(price -> price * 0.9)
  .sorted((price1, price2) -> price1.compareTo(price2))
  .map(price -> "$" + price)
  .forEach(price -> System.out.println(price));
```

More Terminal Operations

- Used to terminate the pipeline
- Some produce a side effect and some return a final result
- Whatever transformations we apply to the streams, the original data source remains unaffected

`toList()` Operation

Terminates the pipeline by returning every element in the stream as a List

```
 List<String> usernames = Arrays.asList("blueEyedDreamer", "FiercePhoenix", "Wildflower87", "SerendipitousSurprise");

List<String> upperUsernames = usernames.stream()
  .map(username -> username.toUpperCase())
  .toList();
```

`reduce()` Operation

Terminates the pipeline by reducing the entire stream to a single value

Example 1

```
 List<Double> earnings = Arrays.asList(40.50, 60.00, 120.50, 20.00, 50.50, 20.00);
Double totalEarnings = earnings.stream()
  .reduce(0.0, (subtotal, element) -> subtotal + element);
// starting point, lambda expression -> BinaryOperator

System.out.println("\nYou've earned $" + totalEarnings); // $ 311.5
```

Example 2

```
List<Double> expenses = Arrays.asList(2.50, 4.00, 5.50, 2.00, 5.50, 2.00);
Double fundsRemaining = expenses.stream()
  .reduce(100.0, (subtotal, element) -> subtotal - element);
// starts from 100$ and keeps reducing w each element

System.out.println("\nYou still have $" + fundsRemaining); // $ 78.5
```

`count()` Operation

Terminates the pipeline by counting number of elements in a stream

```
List<Integer> numbers = Arrays.asList(1, 1, 1, 2, 2, 1, 1, 2, 2, 2, 2, 3, 4, 1);

int count = (int) numbers.stream()
  .filter(number -> number == 1)
  .count();

System.out.println(count);
```

Creating Streams

- Streams can be created from various data sources
- Stream pipelines are not designed to process entire data source at once hence divided into streams

Creating Stream from Datasource: Array

```
String[] greetings = new String[] { "Hello!", "Hola!", "Bonjour!", "Hallo!" };

Arrays.stream(greetings)
  .forEach(greeting -> System.out.println(greeting));
```

- `stream()` methods converts arrays into stream of elements
- resulting stream can be processed by the pipeline one at a time

Creating Stream from Datasource: File

```
try {
  // get the path of the file
  Path path = Paths.get("chorus.txt");
  // convert into stream of string elements (lines)
  Files.lines(path).forEach(line -> System.out.println(line));
}
catch (IOException e) {
  System.out.println("Error: " + e.getMessage());
}
```

- Since the file path can be null by mistake and will throw IO exception hence we need to handle that
- `lines()` method reads all the lines of the passed file into a stream of string elements where each element represents a single line of the file
- basically, each line of the file becomes an element of the stream

Miscellaneous

Package and Import

Package

- `package` keyword specifies the package that a class belongs to

If we are in a parent directory where there's another child directory with some classes, those classes need to specify the package they belong to

```
package model;

public class Game {
  main();
}
```

```
package model;

public class Team {
  main();
}
```

Import

- `import` keyword allows you to use classes that exist in other packages
- Needs to be used when using classes from another package

For eg, if we want to use some classes from the `/model` directory in the `main` directory, we'd need to import it first

```
import model.Game;

public class Main {
  main();
}
```

- Built-in packages are also imported from their parent classes

```
import java.util.HashMap;
import java.util.Map;

public class Main {
  main() {
    Map<String, String> tree =  new HashMap<>();
  }
}
```

Nested Packages and Imports

Let's say a file called `Test.java` is present in `src/main/java/com/dk` directory and needs to be used in the root directory in the `Main.java`

Package

```
package src.main.java.com.dk;

public class Test {
  public void print() {
    System.out.println("Hello World!");
  }
}
```

Import

```
import src.main.java.com.dk.Test;

public class Main {
  main () {
    Test test = new Test();
    test.print();
  }
}
```

> Some IDEs like VSCode have excellent support for these packages and imports and usually update them on their own whenever there is a change.

Static and Final

Static Keyword

- An instance field belongs to objects of the class
- A **static** field belongs to the class

User.java

```
public class User {
  // instance fields
  private String firstName;
  private String lastName;

  // static field
  private static int userCount = 0;

  public User(String fName, String lName) {
    userCount++;
    this.firstName = fName;
    this.lastName = lName;
  }

  // instance methods
  public String getFirstName() {
    return this.firstName;
  }
  public String getLastName() {
    return this.lastName;
  }

  // static method
  public static int getUserCount() {
    return userCount;
  }
}
```

Main.java

```
public class Main {
  public static void main(String[] args) {
    User liam = new User("Liam", "Smith"); // userCount = 1
    User olivia = new User("Olivia", "Davis"); // userCount = 2
    User ethan = new User("Ethan", "Martins"); // userCount = 3
  }
}
```

Printing Instance Field

```
User liam = new User("Liam", "Smith");
System.out.println(liam.getFirstName()); // Liam
```

Printing Static Field

```
// Need to use class since it belongs to class
System.out.println(User.getUserCount); // 1
```

> **Static** keyword means belonging to a class

Static Final CONSTANT

- A **constant** is a field that is same across all objects
- The **final** keyword ensures it cannot be assigned a new value

- A constant field is **static** because it belongs to the class

For example, a minimum working age requirement

```
static final int MIN_AGE = 18;
```

- If we try to assign it a new value, Java will throw errors
- Declared with *UPPERCASE_AND_UNDERSCORES*

> General Rule: Favor constants over loose values

- In case a value is updated, instead of updating it at 10 different places (*loose values*), it's easy to update it at a single place (*constants*)
- Reduce the chances of bugs
- Constants increase readability of code

Some more examples

```
static final int MAX_AGE = 60; // 4 bytes
static final double TAX = 0.13; // 8 bytes
static final int MAX_RETRY = 10; // 4 bytes
```

Accessing Constants

- Since they are **static** and belong to classes, where they can be accessed directly

Employee.java

```
// static final CONSTANT
public static final int MAX_AGE = 65;
public static final int MIN_AGE = 18;

// instance method using `static final CONSTANT`
public void setAge(int age) {
  if (age < MIN_AGE || age > MAX_AGE)
    throw new IllegalArgumentException("INVALID AGE");
  this.age = age;
}
```

- Since they belong to the class, they can only be called by class itself in `main` files

Main.java

```
Employee john = new Employee("John", 45);
Employee lisa = new Employee("Lisa", 20);

System.out.println("Retirement Age: " + Employee.MAX_AGE);
System.out.println("Minimum Working Age: " + Employee.MIN_AGE);
```

Static Imports

- We can import static fields from any classes
- Use normal `import` syntax
- Use `static` keyword to specify a static field is being imported

For example, let's say we are using `Math.PI` at several places which is a **static** field of the `Math` class

Problem

```
public double getCircumference() {
  return 2 * Math.PI * radius;
}

public double getArea() {
  return Math.PI * radius * radius;
}
```

Instead of writing the class name again and again, we can directly import it and get direct access to it

Solution

```
import static java.lang.Math.PI; // specify we're importing a static method

public double getCircumference() {
  return 2 * PI * radius;
}

public double getArea() {
  return PI * radius * radius;
}
```

Now, let's say we have to use several **static** methods of the `Math` class, in that case, it can become a bit tedious to import them all one by one.

So we can import all the static methods from a class at once -

Import all static methods from `Math` class

```
import static java.lang.Math.*;

// PI
public double getCircumference() {
  return 2 * PI * radius;
}

// max
public double getLargerRadius(double otherRadius) {
  return max(radius, otherRadius);
}

// min
public double getSmallerRadius(double otherRadius) {
  return min(radius, otherRadius);
}

// pow
public double getRadiusSquared() {
  return pow(radius, 2);
}

// sqrt
public double getRadiusSquareRoot() {
  return sqrt(radius);
}
```

Enums

- Use **enums** when a variable is limited to a fixed set of values

In this example, the trafficLight can be only among the 3 values - *RED*, *GREEN*, and *YELLOW*.

Example

```
public void drive(String trafficLight) {
  if (!trafficLight.equals("RED") && !trafficLight.equals("GREEN") && !trafficLight.equals("YELLOW")) {
      throw new IllegalArgumentException("TRAFFIC LIGHT CAN ONLY BE RED, GREEN, YELLOW");
  }

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

- Makes code easier to read and understand
- Catch errors at compile-time rather than run-time
- Enums are static by default

- The constants inside it are final

Creating an Enum

- Can be created using the `enum` keyword
- Use *CamelCase* to name the enum
- Values are written inside curly braces `{}` using uppercase and underscore format

```
public enum TrafficLight {
  RED,
  GREEN,
  YELLOW
};
```

Using an Enum

- We can restrict our parameters to be one of the values present in the created enums

```
                //    enum       parameter
public void drive(TrafficLight trafficLight) {
  if (trafficLight == null) {
    throw new IllegalArgumentException("Traffic Light cannot be null");
  }

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

Since the parameters are already restricted to one of the 3 values, we don't need `IllegalArgumentsException` here. We should still have a `null` check since it can always be passed.

Accessing the Enums

- Can be accessed directly from the class

```
import static model.Car.TrafficLight;

public class Main {
  public static void main(String[] args) {
    Car car = new Car("Nissan", 2020);
    // car.drive(Car.TrafficLight.RED);
    car.drive(TrafficLight.RED);
  }
}
```

Here the `TrafficLight` enum will only let us choose 3 possible values - *GREEN*, *RED*, or *YELLOW*.

Big Decimals

- An immutable, used over the `double` type when precision is really really important
- Not all decimals can be represented using `double` type accurately
- Some objects can lose precision with double and cause unexpected behavior while performing arithmetic operations

Issue with `double`

```
double x = 0.1;
double y = 0.2;

System.out.print(x + y); // 0.30000000000000004
```

- `Big Decimal` object can be used to fix such precision issues
- Make sure to use the constructor that expects a *String* to ensure value is represented exactly in the object and the above issue doesn't repeat

Fix with `BigDecimal`

```
BigDecimal x = new BigDecimal("0.1");
BigDecimal y = new BigDecimal("0.2");

System.out.print(x.add(y)); // 0.3
```

When to use Big Decimal?

- In fields such as *data analysis*, *engineering*, *finance*, precision is really important
- Use `BigDecimal` where data is very sensitive and precision matters

Interface

- Contract of Behavior
- Classes with similar behavior should implement an interface
- Every class that implements an interface is required to override all the mentioned methods of the interface
- So if we 2 classes have some similar methods, we can specify them to implement a single interface

Three-Layer Design

1. Presentation
    - Responsible for presenting information to user
    - Direct calls shouldn't be made to repository layer from here
2. Service
    - Middleman between the presentation layer and the repository
    - Contains the business logic of the app
3. Repository
    - Only layer that can access the data store
    - Can perform CRUD operations on the data store

POJO

- Plain Old Java Object
- Represent data that will be saved to a database
- They don't have any special behavior unlike *service layer*

How is Interface different from Inheritance?

- Inheritance allows a subclass to inherit all the features of the parent class while implementing an interface only requires the class to provide an override the methods specified in the interface.
- A class can only inherit from one parent class, but it can implement multiple interfaces.
- In other words, implementing an interface is like agreeing to a contract that specifies certain methods must be implemented, while inheritance is like inheriting all the belongings of a family member.

Conclusion

Thank you for viewing this cheatsheet!

If you found it helpful please check out more of my work on yodkwtf.com or follow me on twitter. I also run a small youtube channel called Yodkwtf Academy.