Java is a programming language AND a platform (any hardware or software environment in which a program runs)

Java is a strong typed language.

IE.variable type cant be adjusted on the fly.no mixing variable types

Java is a static typing language.

IE.must declare variable type when assigning variables

Java is a compiled language.

IE.this means that it goes through analysis and optimization before it is run.

Java virtual Machine (JVM)

- no actual files. Astract machine. Abstract concept

- a group of processes

- provides runtime environment for java bytecode to be executed (.class file)

- loads code

- verifies code

- executes code

- provides runtime environment

Java Runtime Environment (JRE)

- Provides runtime environment

- implementation of JVM

- physically exists

- contains libraries and other files that JVM uses at runtime

Java Development Kit (JDK)

- Physically exists

- Contains JRE + development tools

- contains the compiler

Java runs fast bc it is compiled and converted to bytecode to run on any system os

public class HelloWorld {

public static void main(String[] args) {

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

}}

Must use "" for strings

javac HelloWorld.java

to compile "javac"

Upon success, the Java compiler automatically generates a .class file. This

file contains bytecode.

java HelloWorld

to run "java"

1. HelloWorld is a class. Java uses object-oriented programming for

everything. In fact, it is not even possible to write a function

that doesn't belong to a class. We'll dig in more with Java OOP right after

dealing with data types and loops. It is important that your file

name is exactly the same as your class name.

2. main is a method that defines the entry point for an exectuable Java file;

it is required for any file that is intended to be run from

the command line. A method is just a name for a function that belongs to a

class; since everything in Java must belong to a class, all functions

in Java are methods. Your methods will contain the functionality that you

want your class to have.

3. System.out.println("") is the bit of code that enabled us to output

something to our command prompt or terminal. It is equivalent to Javascript's

console.log() or Python's print().

Syntax

Case Sensitivity: Java is case sensitive, which means "Hello" and "hello"

have a different meaning.

Class Names: Class names are written in PascalCase, also known as

UpperCamelCase. Each word is capitalized, and there are no spaces between

them.

IE.

class MyFirstJavaClass {

}

Method Names: All method names should start with a lower case letter. If

there are multiple words in the method, they should be written in

lowerCamelCase. Typically, we use verbs as method names, rather than noun

statements.

IE.

public void getValues() {

}

Access modifiers

The entry point (for an executable Java file) is the main() method so it is important that you have a main() method.

The method signature for the main() method contains three modifiers:

public: This is known as an access modifier. Any public method we write is accessible from any other class, or package,

or method in our project.

private: only accessible thru the parent class.

default: only need to specify default method in interfaces. only accessible in the same package. ie. child from same package.

protected: gives access to subclass of parent from different package (class, subclass, and different packages)

Main method. Driver method

public static void main()

public: method we write is accessible from any other class or method in our project.

static: means that the method belongs to the class (not an instance). only one exists. and you dont need to instantiate an

instance to access it.

void: indicates that The main() method doesn't return anything.

Variables

The pattern for declaring a variable in Java is type variableName;.

To assign a value to a variable, use variableName = value;

public class Variables {

public static void main(String[] args){

int ourInt; // we can declare a variable without setting its value

ourInt = 400; // we can assign a value to the variable later in our code

double pi = 3.14159265; // we can also declare and assign on the same line

boolean bool = true;

char singleCharacter = 'A';

String multipleCharacters = "ABC";

}

}

Primitive Data Types

Java is not 100% purely OOP language, because of primitive data types

// Here are the most-used primitive types in Java:

int: An int holds integers, like 4 or -51. It is limited to numbers between -2147483648 and 2147483647, or around 2 billion.

long: These are for larger integers up to 2^63, or around 2 quintillion.

boolean: A boolean represents one of two values: true or false.

double: Doubles are for floating point numbers like 3.14159265. We will use double rather than float almost exclusively, because they are far more precise.

char: The char data type i11s used to store a single character, like '$' or 'A'.

// Object Types

In addition to the primitive data types, the Java programming language also has object types. Some examples:

Integer: The Integer class wraps a value of the primitive type int into an object.

Long: The Long class wraps a value of the primitive type long in an object.

Boolean: The Boolean class wraps a value of the primitive type boolean in an object.

Double: The Double class wraps a value of the primitive type double in an object.

Character: The Character class wraps a value of the primitive type char in an object.

String: Represents a sequence of characters, like a Python string.

BigInteger: Represents an integer that can be any size at all.

Notice that primitive types are lowercase, while object types are capitalized.

when comparing "==" primitives, "==" compares values

when comparing objects, "==" compares memory location

Type Size in Bytes Range

byte 1 byte -128 to 127

short 2 bytes -32,768 to 32,767

int 4 bytes -2,147,483,648 to 2,147,483, 647

long 8 bytes -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

float 4 bytes approximately ±3.40282347E+38F

double 8 bytes approximately ±1.79769313486231570E+308

char 2 bytes 0 to 65,536

boolean n/a true or false

Final Keyword

- Final - constant in Java

- Immutable - Can't change these Variables

- Naming Convention - All caps

- Cannot inherit or extend a final class

- String is a final class

Explicitly Casting - Going from bigger cast type to small cast type

Explicit casting is when we declare the type that we want to convert to in our source code. For example:

class Casting {

public static void main(String[] args) {

double d = 35.25;

double dd = 35.99;

casting the double d into a int

int i = (int) d;

casting the double dd into a int

int ii = (int) dd;

System.out.println(i);

System.out.println(ii);

}

}

Going from bigger cast type to small cast type

Implicit Casting

Depending on the context, Java will implicitily cast types for us. For example:

class Casting {

public static void main(String[] args) {

int i = 35;

float f = i;

System.out.println("The number is: " + f);

}

}

Primitive vs Object types

In the previous lesson, we briefly talked about primitive and object types. But when should we use one over another?

Here are a couple of reasons for you:

1. Performance: Object types are an instance of a class. They usually hold data and methods. Therefore, their memory capacity is much

// bigger than their primitive counterpart. Let's look an example:

public static void main(String[] args) {

long start = System.currentTimeMillis();

Integer sum = 0;

for (int i = 0; i < Integer.MAX\_VALUE; i++) {

sum += i;

}

System.out.println("Sum: " + sum);

long end = System.currentTimeMillis();

double total = (double) (end - start) / 1000;

System.out.println("Time of execution: " + total + " seconds");

}

Strings

Strings, like any other object, belong to a class. Instances of the String class are immutable,

- String class is final

so once you create a String, it cannot be modified.

String length = .length()

// Length - Strings have a method to determine their length.

public class StringDemo {

public static void main(String[] args) {

String ninja = "Coding Dojo is Awesome!";

int length = ninja.length();

System.out.println( "String Length is : " + length );

}

}

// Concatenate: A String method that allows two strings to be squashed together. Since each string is immutable,

// this results in a brand new string.

String string1 = "My name is ";

String string2 = "Michael";

String string3 = string1.concat(string2);

System.out.println(string3);

// will output My name is Michael

The '+' sign is also overloaded to provide the same functionality:

"Welcome," + " ninja" + "!"; // displays "Welcome, ninja!"

// Format: Another way of concatenating strings!

String ninja = String.format("Hi %s, you owe me $%.2f !", "Jack", 25.0);

System.out.println(ninja);

// Will print out Hi Jack, you owe me $25.00 !

// Where %s is expecting a string

// And %.2f is expecting a float data type. The value 2 will just place two values to right of the decimal point.

// Trim: The trim() method removes any trailing or leading white spaces from the string.

String sentence = " spaces everywhere! ";

System.out.println(sentence.trim()) // This will output: spaces everywhere!

// Uppercase and Lowercase: The String class provides methods to lowercase and uppercase strings.

String a = "HELLO";

String b = "world";

System.out.println(a.toLowerCase()); // hello

System.out.println(b.toUpperCase()); // WORLD

// Equality: We can compare the equality of a string in two ways. Do they refer to the exact same object,

// or do they have the same exact sequence of characters.

String a = new String("word");

String b = new String("word");

System.out.println(a == b); // false. not the same exact object.

System.out.println(a.equals(b)); // true. same exact characters.

// Method declaration

Methods are just functions that are attached to an object. Each method will have a code body and a method

declaration, which includes the access level, return type, name, and parameter variables.

Method Overloading ( a form of polymorphism, compile time polymorphism. checked at compile time )

// Method overloading is when we call two methods the same name. For example, we might have the following two methods in the same class:

// // Which code would be invoked when we call the fizzBuzz method? It depends on the argument type and the method signature! Due to Java's

// type system, if we try to call fizzBuzz with a string and int, it will know that we are trying to invoke the fizzBuzz method with the

// signature fizzBuzz(String, int). This allows us to handle different cases of method invocation based on our inputs. You may not use

// this in the beginning of your Java journey, but knowing that Java allows this and identifies methods like this is extremely important.

// ...

public String fizzBuzz(int number) {

// ...

public String fizzBuzz(String number, int numberTwo) {

...

//

instanceof check if class type

Array

// Arrays in Java are a fixed-size sequential collection of elements of the same type with a zero-based index. In other words, zero-based index means that

// accessing and setting elements is done by an integer where 0 represents the first item, and so on.

// To use an array, we first need to declare it, initialize and add elements or values to it. We can either do these separately or all at once.

Declaration, initialization, and adding in separate steps:

int[] myArray;

myArray = new int[5]; // Initialization array length

myArray[0] = 4;

myArray[1] = 8;

myArray[2] = 8; for

myArray[3] = 5;

myArray[4] = 9;

// All at once:

int[] myArray = {4, 8, 8, 5, 9};

Delete an object by setting it to null

Note: In Java an array has a fixed size (after initialization), meaning that you cannot add or remove items from an array. In the example above,

we declared that this array would contain five items. You cannot decrease or increase the number of elements in this array after initialization.

Note: What will the array default to if you do not explicitly set every value?

List

to sort List

- Collections.sort(List)

- Use a comparator interface

class Sortbyroll implements Comparator<Student> {

// Used for sorting in ascending order of

// roll number

public int compare(Student a, Student b)

{

return a.rollno - b.rollno;

}

}

ArrayList

- is costly if frequent modifications are required (ie upon removal/deletion, subsequent elements shift to the left)

// There may be times where you will find that you need an array that is not fixed in size. In this case, the util module provides us with an ArrayList.

// There are some slight changes in how you interact with an ArrayList, but it is still a sequential zero-based index collection of elements.

// It is not, however, fixed in size. We can add as many items as we need to! To create and add items to an ArrayList you will need to do the following:

import java.nio.file.OpenOption;

import java.util.ArrayList;

ArrayList<Integer> myArray = new ArrayList<Integer>();

// Let's unpack this a bit:

First, what is this<Integer>business?

This is part of what we call generics, and what you need to know is that it tells our ArrayList of the acceptable types it can hold.

If we did not include this(always include it!), then we could add anything to our ArrayList-strings,integers, and all manner of

things. Generics helps to keep our compiler, and us, from making mistakes that would bite us during runtime.The only major thing

you need to know about generics right now is that you should be using them.

// As far as how to use the ArrayList, you can add and get elements in the

// following manner:

myArray.add(10);

int num = myArray.get(0);

// What if you would like to add elements that are of different types? Well, all

// objects are inherited from the Object class. Therefore,

// we can have our generic as an object and add different types of objects! For

// example:

import java.util.ArrayList;

ArrayList<Object> list = new ArrayList<Object>();

list.add(10);

list.add("Hello");

list.add(new ArrayList<Integer>());

list.add(new Double(12.0)); // adding a Double of value 12.0

System.out.println(list); // [10, "Hello", [], 12.0]

Check out the ArrayList documentation.The most used methods are: add, clear, clone, contains, get, indexOf, isEmpty, remove, size.

Try them out and see what you can do with the ArrayList class.

Note:

The<> are

informally referred

to as the"diamond."

// Loops - For loops and While Loops

// We should use a while loop when we don't know beforehand how many times an

// operation must repeat, but we know we have to repeat until a certain

// condition has turned false. We can think of a for loop as a specialized while

// loop that we should use if we know exactly how many times we have to iterate.

While loop

int i = 0;while(i<7)

{

System.out.println("foo");

i++;

}

For Loop

// The most common for statement you'll see is comprised of 4 key parts: the

// initialization; the termination; the increment; and the body statements.

// Let's put a face to the name:

for(initialization;termination;increment){

body statements

}for(

int i = 0;i<7;i++)

{

System.out.println("bar");

}

String

- concatenating "+" will convert data into string.

- String is final and immutable. concatenating does not edit the original string.

instead you are reassigning the variable (new object created each time).

- String pool (saves space) - place in memory where the jvm stores each unique character sequence that is created.

any matching characters/word references will be pointed to in the String pool

String str = "hello"

String str1 = "hello"

both point to same object in String pool. String pool object will stay in pool even if there

is no reference/pointer to it.

-String pool can fill up memory.

- To create two separate objects, use new keyword

String a = new String("word");

String b = new String("word");

System.out.println(a == b); // false. not the same exact object.

System.out.println(a.equals(b)); // true. same exact characters.

- String Methods: length(), charAt(), substring(start, end), indexOf("char", idx to start looking), trim(), valueOf(),

- Use StringBuffer and StringBuilder are mutable, does not create new string in heap or string loop

- StringBuffer is thread safe, StringBuilder is not

- StringBuilder is faster, bc not thread safe

- has .reverse() method

Regex notes;

- Anchors

^ - indicates a matching string starts with whatever follows this "^" character

Example: '^a' matches 'anna' but not 'banana'

'^The' matches any that starts with 'The'

$ - indicates a matching string ends with whatever precedes this "$" character

Example: 'a$' matches 'anna' and 'banana' but not 'fan'

'end$' matches a string that ends with 'end'

^$ - matches exact string between the '^' and '$' character

Example: '^a$' matches only the letter 'a'

- Quantifiers

\* - matches 0 or more of the character that precedes the '\*'

Example: "abc\*" matches "ab" and "abccc"

+ - matches 1 or more of the character that precedes the "+"

Example: "abc+" matches "abc" and "abcc"

? - matches 0 or 1 of the character that precedes the '?'

Example: "abc?" matches "ab" and "abc"

{n} - matches 'n' number of the character that precedes the {n}

Example: "abc{2}" matches "abcc"

"abc{4}" matches "abcccc"

{n,} - matches 'n' number or more of the character that precedes the {n}

Example: "abc{2,}" matches "abcc" and "abccc"

"abc{4,}" matches "abcccc" and "abcccccc"

{n,n} - matches 'n' number up the other 'n' of the character that precedes the {n,n}

Example: "abc{2,4}" matches "abcc" and "abcccc"

"abc{4,5}" matches "abcccc" and "abccccc"

()\* - matches 0 or more of characters inside the parenthesis

Example: "a(bc)\*" matches "ab" and "abcbc"

(){n,n} matches a 'n' number up the other 'n' of the characters inside parenthesis

Example: "a(bc){2,5}" matches "abcbc" and "abcbcbcbcbc"

- OR operator - "|" or "[]"

(b|c) - matches "b" or "c"

Example: "a(b|c)" matches "ab" and "ac"

# -- Capture variables

# We'll use a regular expression group in the app's urls.py file, and then take the name(s) specified

# in the groups as parameters in the associated functions. A few basic regular expression patterns that will

# come in handy include:

in urlpattterns, urls.py, use 'url()' when using regex in url routing

# example, url(r'^articles/(?P<year>[0-9]{4})/$', views.year\_archive)

use 'path()' for more readable url routing syntax

# path('articles/<int:year>/', views.year\_archive)

[] - matches any value in a range - Regex

Example: '[a-z]' matches 'abc' and 'xyz' but not 'b7' or 'ABC'

\d+ - matches digits with at least one digit

Example: '\d+' matches '8' and '876' but not '5c' or ''

\w+ - matches a string with at least one character (including numbers, excluding special characters)

Example: '\w+' matches '8' and 'abc123' but not 'a!' or ''

{n} - matches n number of repetitions of the preceding pattern

Example: '[0-9]{2}' matches '36' but not '876' or '2'

Special characters that require an escape '\''

[$()|\*+?{\

Enums - enumerated constants

- are a Java language supported wsay to create constant values

- more type safe than variables like String or int

- if used properly, enums help create reliable and robust programs

- elements should be named in all uppercase with words separated by underscore "\_"

- should be ALL CAPS

- Enums create public static final variables - unchangable

Enums in switch cases

Date

- SimpleDateFormat() can be used to format/change how the date is represented

// Length Array .size()

Using the for loop,

we can

easily iterate

over an array,

since we

will always

know its length;for example:

int[] myArray = {1,3,5,7,9,13};

myArray.length

ArrayList<String> dynamicArray = new ArrayList<String>();dynamicArray.add("hello");dynamicArray.add("world");dynamicArray.add("etc");for(

int i = 0;i<dynamicArray.size();i++)

{

System.out.println(dynamicArray.get(i));

}

Enhanced For Loop

// for(int i = 0; i < dynamicArray.size(); i++){

// String name = dynamicArray.get(i);

// System.out.println("hello " + name);

// // other operations using name

// }

// Rather than forcing an awkward declaration like this, the enhanced for loop

// was introduced in Java 5. Translating the above

// code to an enhanced for loop would look like this:

for(String name:dynamicArray)

{

System.out.println("hello " + name);

// other operations using name

}

Switch Statements/Conditions

- good for checking specific numerical values or exact inputs

- dont forget to break in each condition block. will run all cases if no break

- can get performance improvements over if else conditions.

- Switch condition creates key,value pairs

Do While loops

executes its code block at least once before checking the condition

- just use a normal while loop instead of a do while

boolean condition = false;

do {

// code runs once , if condition is met, runs code in this block

} while (condition);

// Collections and Random

Collections do not accept/store primitive data types.

- can store objects, Autobox types

import java.util.ArrayList;

import java.util.Collections;

import java.util.Random;

public class CodeExample {

public static void main() {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(5);

numbers.add(4);

numbers.add(3);

numbers.add(2);

numbers.add(1);

Collections.shuffle(numbers);

System.out.println(numbers); // [1, 5, 2, 4, 3]

Collections.sort(numbers);

System.out.println(numbers); // [1, 2, 3, 4, 5]

Random r = new Random();

System.out.println(r.nextInt()); // without bounds

System.out.println(r.nextInt(5)); // with bounds 0 to 5

}

}

Map

- does not inherit from Collection interface, bc it stores <key,value> pairs.

Hashtable

- threadsafe, whole table is Synchonized

- no null keys or values

Concurrent HashMap

- every entry is synchronized.

HashMap

- not threadsafe, not Synchonized

- allows one null key and multiple null values

- hashmap is faster

HashMap

// Equivalent to Dictionary

// One of the most commonly used map implementations in the Java platform are HashMaps. These allow you to store sets of key value pairs,

// but does not have any order, even if you iterate over it! However, if you wanted to have order, you could use another implementation (TreeMap),

// but the most common Map to use is HashMap since we are usually unconcerned with the order when using a map.

import java.util.HashMap;

HashMap<String, String> userMap = new HashMap<String, String>();

// Once you've created it you can put your key-value pairs into it:

userMap.put("nninja@codingdojo.com", "Nancy Ninja");

userMap.put("ssamurai@codingdojo.com", "Sam Samurai");

// And get them out:

String name = userMap.get("nninja@codingdojo.com");

// Iterating over a HashMap

There are a few ways to iterate over a HashMap. Here, we are going to show you the 2 most common ways to loop over a HashMap

import java.util.Set;

public class HashMapFun {

public static void main(String[] args) {

HashMap<String, String> userMap = new HashMap<String, String>();

userMap.put("nninja@codingdojo.com", "Nancy Ninja");

userMap.put("ssamurai@codingdojo.com", "Sam Samurai");

userMap.put("wwizard@codingdojo.com", "Walter Wizard");

// get the keys by using the keySet method

Set<String> keys = userMap.keySet();

for(String key : keys) {

System.out.println(key);

System.out.println(userMap.get(key));

}

}

}

To iterate over a HashMap, we introduced a new type called Set. In java, a Set is what we call an interface. We will be introducing interfaces in the Java OOP section,

but for now, just think of an interface as any other type. By definition, a Set is a collection that contains no duplicates, which is perfect for a HashMap.

Therefore, the keySet method returns a set of all the keys in our map. set doesn't have indices. Then, in our for-loop, we iterate over the set, print the keys, and get the value in our map with said key.

You can do a quite a bit more with HashMaps. Some of the most common used methods are: clear, containsKey, containsValue,isEmpty, keySet, remove, replace, size, values.

Try these out to see what you can do with HashMaps.

Set

Set<E>

-stores unique elements, no duplicates

TreeSet

- sorts the set

HashSet

- does not sort. applies a hashcode

Throwable

- Exception - meant to be handled.

null pointer Exception

number format exception

arithmeticexception

illegalargumenteception

IOException

- errors - cant necessarily be handled with code

- could be something going wrong outside your code

Exception Handler

- JVM hands exception object to Default Exception Handler

- stops code run

Use Try/catch as exception handler

Try/Catch Try and Catch blocks

class DeliverMessage{

public static void main(String[] args){

UnreliableFriend friend = new UnreliableFriend();

try{

friend.deliverMessage();

System.out.println("The message was delivered!");

} catch (OutOfGasException e){

System.out.println("Hey, uh, so, I ran out of gas..");

// back-up plan here.

}

}

}

finally block

- runs even after returning something in try or exception blocks

- use to close stream, usually file streams

- it will usually execute

- will not run if jvm runs out of memory, or if you unplug your computer before it runs

Throws Exceptions - use if dont want to handle exception with try/catch

Compile vs Runtime

- Exceptions are checked at compile time

- Unchecked or (try/catch) exceptions occur at runtime (try/catch)

OOP - Java

oop pillars

Object Oriented Programming - 4 pillars

Inheritance - is a mechanism in OOP to design two or more entities that are different but share many common features

- features common to all classes are defined in the superclass (a child/subclass class inheriting from a super class or superclass)

Polymorphism - is a mechanism in OOP where one element of code can have many forms

- polymorphism can be implemented in check as objects and class, and class methods

- IE. method overloading (compile time polymorphism), method overriding (runtime polymorphism), ability for object to take on different forms

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

- only need to know surface level. don't need to know how the code works

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

- access to code is shielded

Class Relationships

Composition - a class can contain an instance of another object

- "has a " relationship

- a car has a wheel

Inheritance - class inherits from a super/parent classes

- 'is a ' relationship

- a student is a person

An Object is a collection of behaviors and properties that all revolve around the same concept; this collection of behaviors and properties

are called instance members. Objects were originally used to model real-world objects or situations, but can and are used for all sorts of applications.

They are reusable and are created from blueprints known as classes.

A final class cannot be inherited/extended

Casting to different class types

- Upcast

- Casting from child to parent

- Downcast

- casting from parent to child, not usually, but you can down cast only if object was originally created as the child, upcasted, then downcasted

// Attributes

attributes, to store data about the object

In Java, attributes are referred to as member variables or fields. There are three types of variables in Java, and it is important to differentiate

the different types of variables that can be in a Java class:

Member variables - object attributes or fields.

Local variables - variables in method/code blocks that are not member variables (these are the primary type of variable you have been using so far)

Parameter variables - those that are declared in your method.

Note: You will see member variables called attributes, instance variables, fields, and possibly some other things, these all refer to the same thing.

Constructors and Method Overloading

A constructor method is a special function that gets called when an instance of an object is created. The differences between a

constructor method and the other methods are:

// It doesn't have a return type.

// The name of the constructor MUST be the same as the name of the class to which it belongs.

// A constructor is called automatically when a new instance of an object is created.

Constructor methods are used to execute some block of code on object instantiation. For example, what if we wanted to set the color of a vehicle to a

default instead of calling the setter method after we created the object?

class Vehicle {

private int numberOfWheels;

private String color;

public Vehicle(String color) {

// setting the color attribute to the value from the color parameter

this.color = color;

}

// ...

// getters and setters removed for brevity

// ...

}

this Keyword

Both code samples do exactly the same thing. You can refer to this in any method of your class. There is one case when you MUST use this is

if one of your parameter variables shares the same name as one of your member variables. For example:

class Person {

private int age;

private String name;

public Person(int age, String name) {

this.age = age;

this.name = name;

// this will not set the attribute value. 'age' is a local variable in the constructor method and its value is the very first argument.

age = age

}

}

Constructor Overloading

// One of other uses of the this keyword is when you have overloaded your constructor, but don't want to have to write any repetitive code.

// Imagine that you have written out a constructor that takes an int, a string, and another int, but you also want there to be an empty constructor with default values.

// You could implement such a thing as shown:

public class Person {

private int age;

private int cmHeight;

private String name;

public Person() {

this(20, "John Doe", 171);

}

public Person(int age, String name, int cmHeight) {

this.age = age;

this.name = name;

this.cmHeight = cmHeight;

}

// ...

In this case, it must be the first statement in your constructor, and it will call the other constructor for you.

Object Superclass

Since this refers to your object and given that all objects are descendants of the Object class, this means we are able

to use methods inherited from the Object class.

The Object class is considered a superclass because it is at the top of the class Hierarchy. It contains some methods that

all of our objects can use, but to access them, you have to use the this keyword. Some of the most used methods from the Object class are:

.getClass(): returns a Class object that represents the object's current class.

.equals(): compares two objects for equality and returns a boolean.

.toString(): return a string representation of the object. If you want, you can override this method.

class Person {

private int age;

private String name;

public Person(int ageParam, String nameParam) {

this.age = ageParam;

this.name = nameParam;

}

public void objectMethods(Person anotherObject) {

System.out.println("Class name: " + this.getClass().getSimpleName());

System.out.println("toString: " + this.toString());

System.out.println("Equals: " + this.equals(anotherObject));

}

}

class PersonTest {

public static void main(String[] args) {

Person person1 = new Person(10, "Person1");

Person person2 = new Person(5, "Person2");

person1.objectMethods(person2);

}

}

Output:

Class name: Person

toString: Person@7852e922

Equals: false

Superclass, subclass, and extending

public class Human extends Mammal{

}

In this case, Human would be a subclass of Mammal, and Mammal would be the superclass of Human. The extends keyword tell our compiler of the relationship,

and now the Human class has inherited all the members from its superclass (except it won't have access to private member variables, unless they were exposed

through getters and setters).

class Mammal{

private boolean sleeping = false;

public void regulateTemperature() {

System.out.println("My temperature is just right now.");

}

public void startSleeping() {

sleeping = true;

System.out.println("ZzZz");

}

public boolean isSleeping(){

return sleeping;

class Human extends Mammal {

public void goToWork(){

System.out.println("I'm going to work, something only humans can do.");

}

}

class HumanTest {

public static void main(String[] args) {

Human h = new Human();

h.regulateTemperature();

h.startSleeping();

h.goToWork();

boolean sleeping = h.isSleeping();

if (sleeping){

System.out.println("The human is sleeping!");

}

}

}

Results:

// My temperature is just right now.

// ZzZz

// I'm going to work, something only humans can do.

// The human is sleeping!

Method Overriding (a form of polymorphism, runtime polymorphism)

// You can change or extend the method of a superclass by just writing a method that has the same method signature and return type as the parent method.

// Then the new method will be called on the subclass whenever it is invoked. From our example above, we could overwrite the sleep method for example:

class Human extends Mammal {

// ...

public void startSleeping() {

System.out.println("Toss and turn");

}

}

Now humans will only "toss and turn" when they try to start sleeping. What if we want to give them some rest and just extend our parent class?

Then we can use the super keyword to invoke the superclass method in our own:

class Human extends Mammal {

// ...

public void startSleeping() {

System.out.println("Toss and turn");

super.startSleeping();

}

}

cannot method override a static method

Static Variables and Methods

static keyword to create attributes and methods that belong to the class, instead of the instance of the class.

// An interface is similar to a class in the following ways:

An interface can contain any number of methods.

An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.

The bytecode of an interface appears in a .class file.

Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

// However, an interface is different from a class in several ways, including:

You cannot instantiate an interface.

An interface does not contain any constructors.

An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

An interface is not extended by a class; it is implemented by a class.

An interface can extend multiple interfaces; a class can only extend from one class, but it can implement multiple interfaces.

An interface can have abstract, default, or static methods.

All interface attributes must be public static final

// Interface Methods

There are 3 types of interface methods: default, static and abstract.

New in Java 8, default methods allow developers to create default methods that do not need implementation.

Static methods are interface member methods. Also new to Java 8, they are implemented on the interface level and are called on the interface, not the class.

Therefore, you cannot override static interface methods. You can only call them directly on the interface that they were declared.

Abstract methods are st that are declared without implementation. We saw an example of that in the OperateBicycle example below.

// interface methods are ALL ABSTRACT CLASSES

// therefore we only have methods but no bodies.

// kind of like an ultimate blue print

// Abstract class - kind of like a blue print

abstract classes can have constructors and methods with or without bodies

cannot instantiate an abstract class

can have instance attributes that are not static and final

if you have a method that does not have a body and is abstract then you have to define it in the subclass

- can have abstract methods, implemented methods, and instance attributes

When to use interfaces vs abstract class

- Use interface when you want to take advantage of multiple inheritance. bc a class can only extend one class whether it be an abstract class or not

public interface OperateBicycle {

// constant that is public, static, and final

double myConstant = 3.0;

// default method that does not need implementation

default void sayHello() {

System.out.println("Hello everybody");

}

// static method that does not need implementation

static void staticMethod() {

System.out.println("Hello from the static method of the interface");

}

void speedUp(int increment);

void applyBrakes(int decrement);

}

class Bicycle implements OperateBicycle {

private int speed;

public Bicycle() {

speed = 0;

}

// a static method that calls on the interface static method

public static void staticMethod() {

OperateBicycle.staticMethod();

}

// implementing speedUp

public void speedUp(int increment) {

speed += increment;

}

// implementing applyBrakes

public void applyBrakes(int decrement) {

speed -= decrement;

}

public int getSpeed() {

return speed;

}

}

class BicycleTest {

public static void main(String[] args) {

Bicycle b = new Bicycle();

// print constant

System.out.println(Bicycle.myConstant);

b.sayHello();

Bicycle.staticMethod();

b.speedUp(3);

b.applyBrakes(2);

System.out.println(b.getSpeed());

}

}

What was added in Java 8?

1. Exceptions

- an exception is an interruption in the execution of a program

- Exception thrown - normal flow of a program is terminated (passed on/propagated)

- exception caught - exception is caught (handled immediately)

Throwable Hierarchy

- Errors are problems within the JVM itself

- Exceptions are objects that represent situations that the developer should handle

Uncaught exceptions

- JVM searches call stack for code containing instructions matching the Exception called the Exception Handler

- if no matching handler, JVM hands Exception object to Default Exception Handler

- program will terminate abnormally and print stack trace to console

a.Runtime vs Compile Time Exceptions

- Runtime exceptions - unchecked exceptions

- handling with try/catch or throws keyword is optional. program will terminate at the line of the exception

- Compile time exceptions - checked exceptions

- checked at compile time

- must be handled with a try/catch block or throws keyword

- A try/catch block must be used for exception handling

➔ The code within try block executes normally until exception occurs

➔ Remaining code in try block skipped, code in catch block immediately executed

- Multiple Catch

➔ A single try block can be paired with multiple catch blocks

➔ Each catch block will handle a different kind of exception

➔ A more specific kind of exception cannot follow a more general exception

Finally Block

➔ Cases where code should be executed whether an Exception occurs or not

➔ Code within a finally block is always executed

◆ Exception: program exits by System.exit() or by fatal error or hardware malfunction

Try With Resources

➔ A try-with-resources block automatically closes resources opened within the try

➔ Required resources must be “passed” into try block like a parameter

➔ Added in Java 7

Throwing Exceptions

➔ Exceptions can be thrown directly via throw keyword

➔ Are objects so can be instantiated with the new keyword

➔ Have a constructor with string parameter

◆ Sets the message that describes the exception thrown

Propagation

➔ Exceptions don’t have to be handled in method they occur

➔ The throws keyword in method header propagates exception up call stack

b. Custom vs Built-In Exceptions

- Custom exceptions

programmer defined exceptions can contain custom data members to give more details about an exception

- created by extending the Exception Class

2. Testing and JUnit

- Assertions

- great for debugging and during development. great for pinpointing errors

- assert keyword is used to check if a condition is true

◆ used on conditions that must be true if code is working correctly

If assertion false, AssertionError thrown and program stops/terminates

- Message may be displayed in the console after assertion using following syntax:

assert condition : "Condition is false";

➔ By default, Java skips assertions on Runtime

3. Files

The File class is from the java.io package and allows us to handle files as objects in our program

➔ Can’t modify/access content directly with just File

➔ Why use it?

◆ Different platforms have different file structures

◆ Can handle abstract representation of files and

directories

◆ Used to rename and delete files

◆ Create directories

a. File Object

b. Reading/Writing and Characters vs. Bytes

- FileReader - will read a series of characters from a source

To read/write to files, must create stream objects and attach them to a file

➔ Streams are a sequential flow of data items

◆ Input streams have data coming in (read)

◆ Output streams are sending data out (write)

➔ Two types of streams

◆ Character Streams - character data types

◆ Byte Streams - raw bytes (not just character data)

c. Reading/Writing Objects and the Serializable Interface

ObjectOutputStream and ObjectInputStream can be used to save and load objects from a file

➔ If you wish to save objects from a class, that class must implement the Serializable interface

-> ObjectOutputStream can write primitives, collections and objects to a file

- transient keyword - things labeled transient will not be written to the file.

d. Buffered Reading/Writing Classes

- BufferedReader buffers characters to enable efficient reading of text data

- Wrapper over Reader class

- Minimizes I/O operations by reading chunks of characters and storing them in an internal buffer

- BufferedWriter can write large chunks of data to a file at once without delayed performance

- Also uses buffer and more efficient way of writing to file

- Wrapper over Writer class

4. Threads

- Threads allow for a program to run tasks in parallel

- Most cases, threads should be synchronized

◆ That way threads do not handle same data in

memory concurrently

◆ Thread will wait till another thread finishes before proceeding

➔ Can be created using Thread Class or Runnable Interface

a. Thread Class vs Runnable Interface - generally preferred to use Runnable interface,

if using thread class, every thread creates a unique object, where as with the thread interface, threads share the same object

Thread Class

- Can create a thread by extending the Thread class

➔ Provides constructors and methods to create and perform operations on a thread

➔ Call start() method to start thread and execute code from run() (super.run())

- override the run method

Runnable interface

Create thread also by implementing Runnable Interface

➔ Allows for multiple inheritance

➔ Cannot start thread directly with

- have to provide implementation for all methods

Runnable, must pass object into to a new Thread object

b. Run method vs calling .start()

c. Thread Lifecycle (new, runnable, running, non-runnable, and dead)

- new - thread created, has not started

- runnable - call start() on new thread, not doing anything, but in a ready state, lock removed

- non-runnable - thread alive, but not running, waiting for other thread to finish or for resources to free up

- was in sleep(), suspend(), await(), return

- dead - finished executing

d. Synchronized methods and blocks

- synchronize as little as possible and be as specific as possible

- Synchronization needed to control thread coordination

➔ Especially to prevent simultaneous operations on data

➔ Synchronize entire method:

- only one thread can access a synchronized object at a time

- synchonized(this) - the instance of an object

- synchonized(object) - reference of an object

e. Inter-Thread Communication (rest, wait, notify, notifyAll)

- sleep() - pauses the thread from executing

- wait() - tells calling thread to release lock and sleep until another thread calls notify() or notifyAll()

f. Deadlock - two or more threads blocked forever, and waiting for each other

- charge order of threads lock and access resources

- provide the proper sequence of locks and unlocks

- different threads should lock multiple resources the same order

Java ***inner class*** or ***nested class*** is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

1. Can have ***access to all members*** (data and methods) of outer class ***even if private***
2. More ***readable*** and ***maintainable code***

a. Logically ***groups classes*** in a

single block. 3. ***Code Optimization***

a. Less code to write.

5. Singleton Class

- Singleton is a class that can only have one object (instance) at a time.

- Trying to instantiate a Singleton Class after one has already been created returns a pointer

to the already existing Class Instance (object)

- 3 components to singleton class

Static member : This contains the instance of the singleton class.

Private constructor : This will prevent anybody else to instantiate the Singleton class.

Static public method : This provides the global point of access to the Singleton object and returns the instance to the client calling class.

- great for controlling/managing a connect to a database

6. Annotations

 - a form of metadata, provide data about a program that is not part of the program itself.

- Use the @ sign followed by keyword text

● Are markup tools that indicate to a compiler, dev tool, deployment tool,

runtime environment, or framework that binds to the associated code

● @Override and @FunctionalInterface are two common annotations

- Three Types of Annotations

● Marker

● Single-Element

● Full Value or Multi Value

7. Cloneable

a. Reference vs Clone

- When making new Objects and using the ‘=’ operator

● This will only create a new reference to the same memory location

● So changes of the contents in memory will reflect all variables the reference that object

b. Cloneable Interface

- Is a Marker Interface

● Has no content

- Object has default implementation for .clone() method

● Classes must use Cloneable to enable .clone() from Object

● .clone() can be overriden once Cloneable is implemented

Default .clone() makes copies of all primitive data fields of a class

c. Shallow Clone vs. Deep Clone

Shallow Clone - Is the default implementation of .clone() from Object

● Only primitive data of class is made into new copy

Deep Clone

- Must Override .clone() so that composed objects are also copied, and not passed by reference.

● Otherwise changes to the original object will affect the clone

8. Lambda Expressions

a. Functional Interface

- Interface that has only one abstract method

- An Example is the Runnable Interface

- Implemented with Lambda expressions, part of updates in Java 8

- Java provides an annotation @FunctionalInterface, which can be used to declare a functional interface.

b. Lambda Expression syntax and advantages

- Lambda expressions provide implementation of functional interfaces.

- Java lambda expression is treated as a function, so compiler does not create .class file.

- Provides a clear and concise way to represent one method interface using an expression.

9. Streams

a. Stream use and advantages

- can create streams from array - Stream.of(array);

- create from collections - list.stream();

- create by Stream.generate() and Stream.iterate()

- or .stream() off of a collection object

- Arrays.stream(array)

- Stream.of("djkfd", "akdsjfds", "ksljfds")

- NOT data structures but are wrappers around Collection that carry values from a source and process

them through a pipeline of operations.

- Streams are a way to implement “functional” (style) programming in java (object oriented)

Are parallelizable

○ Can be “on-the-fly”

○ Are designed for lambdas

○ Can easily be output as arrays or lists

○ Employ lazy evaluation

○ Are called sequential streams

-

b. Intermediate methods vs Terminal Methods, and Short Circuit methods

- Intermediate - returns a stream

map() - returns a stream consisting of the results of applying the given function to the elements of this stream.

can convert to different type

filter() - returns a stream consisting of elements of the stream that match the given predicate

distinct, sorted, peek, limit, parallel

- Terminal - returns a non stream object - primitive, object or nothing

forEach() - void

toArray() - returns an array containing the elements of this stream

reduce() - returns one single value of same type of object as a result of a function applied to the object type

collect() - returns a collection

min() - returns the minimum element of the stream based on the provided Comparator. A Comparator is a comparison

function, which imposes a total ordering on some collection of objects. min() is a terminal operation which combines

stream elements and returns a summary result. So, min() is a special case of reduction. The method returns Optional instance.

max() - Stream.max() returns the maximum element of the stream based on the provided Comparator. A Comparator is a comparison function,

which imposes a total ordering on some collection of objects. max() is a terminal operation which combines stream elements and

returns a summary result. So, max() is a special case of reduction. The method returns Optional instance.

count() - long count() returns the count of elements in the stream. This is a special case of a reduction (A reduction operation takes

a sequence of input elements and combines them into a single summary result by repeated application of a combining operation).

This is a terminal operation i.e, it may traverse the stream to produce a result or a side-effect. After the terminal operation

is performed, the stream pipeline is considered consumed, and can no longer be used.

anyMatch() - returns whether any elements of this stream match the provided predicate. It may not evaluate the predicate

on all elements if not necessary for determining the result. This is a short-circuiting terminal operation.

allMatch() - returns whether all elements of this stream match the provided predicate. It may not evaluate the

predicate on all elements if not necessary for determining the result. This is a short-circuiting terminal operation.

noneMatch() - returns whether no elements of this stream match the provided predicate. It may not evaluate the predicate

on all elements if not necessary for determining the result. This is a short-circuiting terminal operation.

findFirst() - returns an Optional (a container object which may or may not contain a non-null value) describing the first element

of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned.

findAny() - returns an Optional (a container object which may or may not contain a non-null value)

describing some element of the stream, or an empty Optional if the stream is empty.

iterator() - method allows us to iterate stream elements till the specified condition. This method returns a sequential ordered

Stream produced by iterative application of the given next function to an initial element, conditioned on satisfying hasNext

predicate passed as parameter. The stream terminates as soon as the hasNext predicate returns false. The resulting sequence

returned by this method may be empty if passed predicate does not hold on the seed value. Otherwise, the first element will

be the supplied seed value, the next element will be the result of applying the next function to the seed value, and so on

iteratively until the hasNext predicate indicates that the stream should terminate.

- Short-Circuit - anyMatch, allMatch, noneMatch, findFirst, findAny,limit

c. Creating Streams and Object Streams vs. Primitive Streams

- The Optional class, wraps an object, and provides protections in the event of NULL being returned - for some terminal streams

d. Common Stream Methods (map, reduce, forEach, filter, ect.)

e. Parallel Streams vs. Sequential Streams

Parallel streams

- Allow for program architects to take advantage of multi - core architectures

By using streams in parallel, some processes of the pipeline can have faster processing time as opposed to sequential streams (regular streams).

Parallel Streams partition the data pipeline into sections for multi thread processing.

When to use:

● Output is NOT dependent on ordering of data

● Processing aggregate functions

● Over large data sets

When NOT to use:

● With unsynchronized share resources

○ Parallel stream creates threads

● When performance CANNOT process multi - threads

3 types of java comments

- single line

// single line comment

- multiline

/\* Multiline

\* this is multiline comment

\*/

- Javadocs

/\*\*

\* This is a javadoc comment

\* used for java code documentation.

\* used to document your code

\*/

Why use Java 11?

- Streams and lambda functions (introduced in java 8) have been improved

- how is anonymous different from lambda function?

- Try with private resources (added in java 7). auto close resources

What is an API?

Application programming interface

REST - representational state transfer

JavaBeans

// A javabean must have a public, no- argument constructor ( a default constructor).

// The javabean class attributes must be accessed via accessor and mutator methods that follow a standard naming convention ( getters and setters and boolean attributes)

// The javabean clas should implement the Serializable interface.

javabean class attributes must be private.