Java

May 2023

Notes

Uriel K Alistair

Contents

[Part 1: Introduction 3](#_Toc145500769)

[1. Control Flow 3](#_Toc145500770)

[2. DataTypes 4](#_Toc145500771)

[3. Basic Syntax 6](#_Toc145500772)

[4. Inputs and Outputs 6](#_Toc145500773)

[Part 2: Classes 7](#_Toc145500774)

[1. Intro: Privacy 7](#_Toc145500775)

[2. Syntax 7](#_Toc145500776)

[3. Subclassing 9](#_Toc145500777)

[4. Polymorphism 10](#_Toc145500778)

[Summary 14](#_Toc145500779)

[Part 3: Interfaces 16](#_Toc145500780)

[1. Interfaces 16](#_Toc145500781)

[2. Iron Fists: Confining code within a class 17](#_Toc145500782)

[3. Callbacks 19](#_Toc145500783)

[4. Iterators 21](#_Toc145500784)

[Summary 23](#_Toc145500785)

[Part 4: Generics 25](#_Toc145500786)

[Summary 32](#_Toc145500787)

[Part 5: Dealing with things at Runtime 33](#_Toc145500788)

[Reflection 33](#_Toc145500789)

[Errors and Logs 34](#_Toc145500790)

[Summary 39](#_Toc145500791)

[Part 6: Collections & Maps 40](#_Toc145500792)

[1. List<E> 41](#_Toc145500793)

[2. Queue<E> 44](#_Toc145500794)

[3. Set<E> 46](#_Toc145500795)

[Maps 46](#_Toc145500796)

[Summary 49](#_Toc145500797)

[Part 7: Features For Flair 51](#_Toc145500798)

[Cloning 51](#_Toc145500799)

[Type Inference 52](#_Toc145500800)

[Lambda Functions 53](#_Toc145500801)

[Part 8: Streams & Optional 54](#_Toc145500802)

[The Optional Class 60](#_Toc145500803)

[Part 9: Concurrent Programming 61](#_Toc145500804)

[Part 10: Farewell 67](#_Toc145500805)

# Part 1: Introduction

As a person who began out of Python, Java is weird. Honestly, my first impression of Java is that it’s an irritating and unintuitive way to code. Simply, a step down from the freedom of C++.

I hope that that impression will change and Java becomes best girl before this season ends.

Anyway, Let’s start getting familiar with this language.

I advise that you occasionally check the Summary of each section as you read the section, if it gets confusing or boring. It helps condense, clarify and establish the feeling that things are under control, and you understand it. The section flow is all over the place at times, but the overarching flow and Key takeaways are all in the Summaries.

### 1. Control Flow

Python is an interpreted language, meaning it just runs through your code line by line. In other languages like C++, which is compiled, there is no such thing as code “floating” in the abyss. Everything is in a function. That’s a bit weird when you think of it that way, but I never even thought about it.

Why? Because the main function really just felt like the basic sandbox where we write random code. Since all code is in either classes or functions, somebody has to call them. Without main, we would never start.

Although the idea that everything must be packaged in functions may feel a bit off, as long as we have a starting point for the code, we can live happily ever after.

Java, however, was built with Object Oriented Programming in mind. It takes this packaging business to 100, and declares the idea of a function “floating” in the abyss unacceptable. **Everything** must be in a class. What absurdity is this?

See, this is the idea behind OOP: The epicentre of our attention belongs to data, not functions. If you’re making a banking system, you have customer data, manager data etc. There’s never any data in isolation. Data is always about somebody or something, and what we want to do really, is to get something done upon this data. Therefore, it makes the most sense to package functions with classes, whose data they will act upon.

Once again, like C++, there arises the question, if everything is in a class, who is creating objects and calling methods? Answer: main.

Here’s how you say Hello world in python:

print("Hello World")

Here’s how it’s done in C++:

#include <iostream>  
  
int main() {  
 std::cout << "Hello World!" << endl;  
 return 0;  
}

And here’s how Java does it:

public class HelloWorld {  
 public static void main(String[] args)  
 {  
 System.*out*.println("Hello, World!");  
 }  
}

Sigh.

It’s clearly got a *lot* more dressing but it’s fairly straightforward. It gets very classy. Looking at the definition line of the main function, we see 3 keywords:

* The void modifier tells that the main function returns nothing.
* The static modifier tells that the function exists and can be called even without an object instance. This is the same as @staticmethod in Python.
* The public modifier means that the method is not an internal construct, and is accessible to some other code trying to invoke it.

Each Java class is stored in a separate file, with the exact name of the class. The code here would go in HelloWorld.java (Case sensitive). If we begin execution in this file, it will call the main method, and start executing. This method may invoke objects from other classes, and Java will go and look for a file with the name of the class.

Side note: Java code is compiled by Javac (java compiler) into filename.class a bytecode file. This bytecode is for the Java Virtual Machine and can be ported to any OS or architecture because the JVM deals with the underlying machine level implementation later. You “write once, run everywhere”.

Inside the method, we see that print has been called, but with some dressing. The print statement is not a direct function, because, in an OOP language, there is nothing that exists without a Class. Here, the System Class has a class variable called out, which is an instance of the class PrintStream, and is mapped to the stdout stream by default. The method println() of the “out” object out is being invoked here. Actually, the PrintStream class has a ton of overloaded functions by the name of print and println so that any datatype can be printed.

That’s almost it. The main function typically runs with no arguments, because nobody is calling it, but we seem to be taking inputs in the form of a string array. That’s because the inputs are for when the code is called using the command line interface. All the options after the core command show up here as args.

### 2. DataTypes

Strictly speaking, for an OOP, there exists nothing apart from classes in the world. There should be no such thing as datatypes. But, of course, that’s a pain. Java gives in here, and allows for 8 static primitive scalar datatypes.

| **Type** | **Size in Bytes** |
| --- | --- |
| int | 4 |
| long | 8 |
| short | 2 |
| byte | 1 |
| float | 4 |
| double | 8 |
| char | 2 |
| boolean | 1 |

The size in bytes is fixed by the JVM. No architecture dependence. (Unlike C++) Each char takes up 2 bytes instead of one because unicode was already being adopted at this point. A 4 digit hex code can represent a character. (Hex = upto 16; Each digit is 4 bits of data, so 4\*4 =16 bits)

char x='\\u03C0'; // Greek letter Pi

Like C++, single quotes are reserved only for characters, and double quotes for Strings. Strings are a Class, we will get to it later.

final float pi = 3.14159265f;

The final keyword protects a variable from being changed. If it is reassigned, an error is thrown. It’s equivalent to const in JS.

When a floating point number is presented to Java, it is automatically assumed to be a double. To insist to the Java compiler that the value is a 4 byte float, use a trailing f. Otherwise, the RHS value is taken to be a double and a “narrowing conversion” or “downcasting” is done, which could be problematic, which is why the compiler won’t allow it.

float f = 22/7 // Results in 3.0, a float output rather than int.

Java has 5 basic arithmetic operators: +, -, \*, / and %.

There is no separate integer division, like in Python, with its //. If two integers are divided, only integer division is performed. If one or more of the arguments is a float, only then float division is performed.

Here, 22/7 gives the integer 3, which is then implicitly typecast into a float, and the compiler wont cry about it since it’s not a narrowing conversion. Any int can be represented with a 4 byte float.

boolean x=true, y=false;

Note the lack of capitalisation.

With primitive data types out of the way, the classes:

* Strings are immutable; Cannot do s[4] to access n th character; It’s not an Array. Use dedicated methods to screw with them.

String s = "Hello";

* Arrays are of fixed size.

int[] x; // alt: can use int x[]; Doesn't make anything or allocate memory.  
x = new int[100];

Note: x.length gives the length of an array. It’s dotted access of an instance variable. s.length() is a method that requires brackets; slight inconsistency that could lead to mild irritation. Remember that strings come with tons of methods and you’ll be good.

### 3. Basic Syntax

It’s pretty much the same as cpp.

* Semicolon as expression delimiter. Braces.
* if else else if, braces
* while, do while.
* for - 2 types: 1. Classic for, identical to while in functionality. 2. for(int x : arr){}

it’s possible to use a predeclared i for the classic for, but the second version only works with a locally scoped variable.

* switch case, no expressions, only constant cases; it flows down without stopping if not specified, so break needs to be used after each case.

### 4. Inputs and Outputs

* The Console class has 2 methods, readLine and readPassword to take inputs. (Note: methods follow camelCase, while PascalCase is used for class names.) Both these methods take one argument which is the prompt shown to the user. readLine takes one line of input and delivers it as a string, like python. readPassword is identical except for the fact that it does not show what the user is typing, keeping it as \* or maybe simply invisible text like most CLIs. readPassword returns an array of characters, not a String.

import java.io.Console;

Console cons = System.*console*();  
String test1 = cons.readLine();  
char[] test2 = cons.readPassword();

* Because a string needs to be split and messed with, it’s a bit of an inconvenience. The more powerful Scanner class can be used to parse the input stream in a more elegant way.

import java.util.Scanner;

Scanner s = new Scanner(System.*in*);  
float x = s.nextFloat();  
BigInteger y = s.nextBigInteger();  
String name = s.next();

The creation of the new scanner object maps the system object’s stdin stream into the Scanner object s.

# Part 2: Classes

### 1. Intro: Privacy

As noted in my doc on Python classes (which also doubles as a direct motivator for Java), variables and methods should have some level of privacy/protection. Rather than permitting direct edits of attributes, there must be well-defined getter and setter methods that will persist as an API even if underlying implementations change.

Python circumvents this issue by creating a pseudo-protected attribute with \_ \_method, and provides an alternate getter and setter functionality accessible via the usual dotted access.

Java is a much more strict language, and sensibly so. It’s python which is the deviant. This is why Java needs a ton of specifier keywords before everything. It prevents errors with rigour and catches issues far in advance, instead of relying on programmer discipline.

### 2. Syntax

* Class definitions begin with an access modifier. The modifier can be omitted, and Java will assume the class visibility to be set to “package”. The modifers that can be specified are private, public and protected. We will worry about these later.
* The keyword “this”, replaces “self” of python, and is used to access object variables. If the fact that it’s a self parameter is obvious, the keyword “this” can be omitted completely. The methods do ***not*** use “this” as an argument, unlike python, which uses it as almost a cosmetic redundancy.

Although at first, I did ***not*** like the idea of not using “this” for object variables, I soon realised it’s really easy to grasp what’s happening. Class variables in python are characterised by not having a self tag, but in Java, they need to be explicitly declared to be so, and it class and object variables are rarely confused.

* Class variables are denoted using the keyword static. Recall that the main method was modified with static to denote that it transcends individual objects and doesn’t need one to exist to be called. Being class variables, despite the name “static” they can be changed.
* Privacy exists for all methods and variables.

public class Employee {  
 int salary;  
 String name;  
 private int secret=42;  
 final static public int myno=666;  
  
 private void secretPrint(){  
 System.*out*.println("My secret Print");  
 }  
  
 public void openPrint(){  
 System.*out*.println("Public printing");  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Employee e = new Employee();  
 System.*out*.println(e.salary);  
 System.*out*.println(e.name);  
 System.*out*.println(e.secret);  
 System.*out*.println(e.myno);  
  
 e.openPrint();  
 e.secretPrint();  
 }  
}

1) Any Variable declared inside a class without access modifiers becomes by default, a package level instance variable. It can be accessed via dotted access of instance objects.

2) A public static (class) variable can also be accessed with dotted access, but it is bad practice to do so on the object, as it is a *class* variable. (Eg: Use Employee.myno instead of e.myno) A private static variable is typically used as a means of sharing some editable common information between all objects of a class, for example, an index assigned to each object that counts up.

3) A private variable cannot be accessed via dotted access outside a class. Attempting to do so will throw an error.

4) A private method cannot be called outside a class. A public method can.

5) A final variable cannot be redefined. A final method cannot be overridden by a subclass.

* Python uses the function \_\_init\_\_ as a constructor, and it is mandatory for all classes to have one. Java demands that constructors use the same name as the name of the class (rather than init). Multiple constructors can be defined (all with the same name), but differing input arguments (called the signature), and the appropriate one will be called. This is called overloading, something not allowed at all in python.

class MyClass { // Implicit access modifier to package level  
 int value;  
 final int answer = 424242; // Class variable

// default constructor  
 public MyClass() {  
 value = 42; // implicit usage of this  
 }  
  
 // parameterized constructor  
 public MyClass(int v) {  
 value = v;  
 }  
}

* Unlike python, it’s possible to not define a constructor at all; A default dummy constructor will set all instance variable values to 0, false, null etc. Even if you have *one* constructor defined though, this behaviour stops and you will have to manually define a default constructor. Also, observe that no return type is specified for a constructor; It’s an abnormal method that **cannot** be called by doing something like x.MyClass().
* A copy constructor takes in another object of the same class, and uses that object’s values for initiation. Here, a dotted access is permitted, **even for private attributes**. (not just here, any method that works with another instance of the same class) Why? It’s between two objects of the same class. Privacy concerns work only for external people messing with how your code should be handled. There are no such concerns inside the class itself.

public Employee(Employee e){  
 salary= e.salary;  
 secret = e.secret;  
}

* Accessors and Mutators (AKA getters and setters) need to be set up to access object variables. They are regular methods, no special syntax or anything.

### 3. Subclassing

Consider the set of Natural numbers. They have some properties. Now, even numbers are a subset of Natural numbers, so all properties of natural numbers are applicable to even numbers too. But, on top of that, even numbers have some more properties. Therefore, a subclass is a subset of the members of a class, but its properties and abilities are a superset of that of the parent class.

The idea of class inheritance enables us to write compact and very cool code, and often, allows us to tailor the abilities of a class somebody else made to suit our needs. The natural questions that follow are: 1. How to syntactically express class hierarchy? 2. How does privacy work across subclasses? And 3. How does a subclass behave in code, and how do we use them elegantly?

public class Employee {  
 protected int salary;  
 private int secret=42;  
 final public int myno=666;  
  
 private void secretPrint(){  
 System.*out*.println("My secret Print");  
 }  
}  
  
class Manager extends Employee{  
 private void getPaid(){  
 System.*out*.println("Got paid"+salary);  
 }  
 public void spillBeans(){  
 secretPrint();  
 System.*out*.println(secret);  
 }   
}

Observe that although a manager is a very cool employee, the manager specific methods cannot use the private variable secret. Further, it’s not possible to invoke the superclass’ private methods either.

This makes sense, because If I’m using somebody else’s code and extending it, I should not be messing directly with sensitive parameters. If they intended for the variable to get visible or editable, they would’ve written getters and setters. If a method had utility outside their own class, they would not keep it private for internal implementation.

That said, it often happens that a function must not be called outside the class definition ever, like a private function, but subclassing becomes inconvenient if the function cannot be called. That’s what the protected modifier is for. In the above example, salary is protected, so random code cannot do e.salary. However, a subclass of employee *can* use the variable salary.

That said, private variables of the employee class might get assigned values in the constructor. When we design a Manager object, we need a new constructor, and this might have to set values to private variables. This is not a concern. It’s possible to simply invoke the superclass constructor in the new constructor. You ***cannot*** directly touch private methods or variables. You can only invoke accessible methods to get what you need done.

When a subclass is defined, it is necessary that the subclass’ constructor calls at least one constructor of its superclass. Since this is true of every class, every constructor chains all the way upto Object. If a manual call to super(args) is not made, super(), the no argument default constructor is invoked at the start of the constructor.

If a class exists with a non default constructor then the default null setting constructor the compiler creates is disabled. This means that the subclass will be unable to call super(). Thus, if a class is subclassed, it must necessarily have either no constructors at all, or have a bunch of constructors, along with a default constructor.

That pretty much covers what needs to be done to define a subclass. (There’s still overriding and a bunch of other stuff, but we will get to it sometime later) Now, I’d like to point out something very cool about java objects.

### 4. Polymorphism

Employee e = new Manager();

That’s a valid declaration in Java. That’s kind of like

NaturalNumber n = new EvenNumber();

The type is declared to be NaturalNumber, but the exact class we end up using is EvenNumber, so it’s not untrue that it’s also a NaturalNumber. This declaration is using a *Subtype.*

Why is this useful? Well, I can put 1000 employee objects in a list, and call eList[i].work()

And if the particular employee object under consideration is a manager, they do a manager’s work, if they’re a CEO, they do their job etc. If a single function is overriden in the subclasses, the function of the most specific class is used when the method is called. Since the method to be invoked is chosen depending on the particular object, despite all of the objects being statically typed as “Employee”, this is called “Dynamic Dispatch”.

This is very different from Static invocation of overloaded methods:

class Arrays{  
   
 public static void sort(double arr[]){...}  
 public static void sort(int arr[]){...}  
}

int[] a = new int[100];  
double[] b= new double[100];

Arrays.sort(a)

Arrays.sort(b)

Here, a very similar thing is happening: Although the same line of code is executed, Different functions are called based on the static type of the object acting on the function.

Dynamic dispatch/polymorphism does not care that all objects have static type employee; It’s decided at runtime only, depending on the exact object.

Employee e = new Manager();

* Object e can call any method of employee.
* If the method is overriden by Manager, then that method is invoked at runtime instead.
* Calling a method of Manager, that does not exist in employee is strictly disallowed by the static type checking of the compiler. This means that e needs to be *typecast* before such a method can be invoked.

if(e instanceof Manager){  
 ((Manager) e).doManaging();

}

Such Typecasting is not just for classes; It can be done for the primitive datatypes seen earlier too. This legalises narrowing conversions, but please do check docs for the exact behaviour of the intended conversion. For instance:

double d = 29.88;  
int newd = (int) d;   
System.*out*.println(newd); // yeilds 29, not 30.

So far, we have seen 2 types of polymorphism (lit. “Many forms”) : 1. The trick with overloaded functions, which is never really called as polymorphism by anyone in the programming sphere, and 2. The default dynamic dispatch behaviour, which can be called *inheritance polymorphism,* but is simply called “polymorphism” by people who use functional programming languages, where the class hierarchy gives this feature. There’s a 3rd type of polymorphism called Structural polymorphism, which is also simply called “polymorphism” by people who use object oriented programming languages.

The idea behind structural polymorphism is this: If I give you bunch of boxes, and say, reverse their order, do you need to look into the boxes? No, you can do it for some general box. If I say sort the boxes, then what? You do need to look in, but what will you look for? Something, right?   
That functionality is called comparability.

One can define a single sort function which sorts on the basis of comparisons between entries. The sort function can simply check if these objects have a comparison method and if they do, the sort works regardless of what data type the sort function is acting on.

Structural Polymorphism is defining generalised functions that set clear boundaries on how far inside the box they must look. Since as long as this structure exists, they do not care about the rest of the box, the functions can work on a very broad range of objects, and thus the function is adopting “many forms”.

Let’s see how we can implement a simple function: find. We will iterate through a list, and find if a particular box is there. How do you know? You need to have an equals method. If the box you need is equal to the box you are looking at then, you return its position.

In python, for instance, when a >= b is invoked where a and b are objects of a some datatype, the \_\_ge\_\_ dunder method is called. This way, you can have a list of specialised objects, and yet use a standard sorting algorithm, because the > and < signs in the code of some stranger actually adapt to the equality operations you define in your class.

We will now attempt to do the same thing in Java. The == sign, always refers to pointer equality, meaning the two objects being compared are literally the same object.

However, this equality is very rarely useful when dealing with complex classes. There is a function called equals, which all classes have, inherited from the class Object. All Classes are subclasses of the Class called Object, which is basically God, the father of all. ***This*** equals function also returns the same result as ==, because that’s just what is done inside it.

The class Object gifts all its children with 2 public methods: equals and toString (and some others, but not a concern atm). Equals is straightforward. toString is like \_\_str\_\_ in python, the idea is to output a single string that tells you about all the object variables and such. The toString function in Object will only return a pointer location as a string with the class name so it’s not of much use.

(x==y) does **not** implicitly invoke x.equals(y); Operator overloading is **not available** in java.

System.out.println(“stringy”+x) implicitly invokes x.toString()

Let’s try to create a function equals for the date class.

public boolean equals(Date d){  
 return ((this.day == d.day) &&  
 (this.month == d.month) &&  
 (this.year == d.year));  
  
}

Now, we write a generalised function we define to find the position at which a given object is present in a list.

public int find(Object[] objarr, Object o) {  
 int i;  
 for (i = 0; i < objarr.length; i++) {  
 if (o.equals(objarr[i])) {  
 return i;  
 }  
 }  
 return (-1);  
}

Now, we run into some trouble. We did something wrong.

When we defined the equals function, we did **not match the signature** of the function to the original object class’ equals function. The original function in Object was public boolean equals(Object o)

The equals we made does not have a matching argument type, so we failed to override the original equals function.

Is that an issue? Not really. But there is a lot of nuance involved in getting away without overriding the real function.

What **is** an issue is that when an object of static type object is passed to equals, the dynamic polymorphism cannot kick in, because fundamentally, a different function is being invoked, the function with the object signature.

This is the case when we try to call find on an array of date objects. The thing won’t work. (Also, yes, you can call find directly on a date array because the expected arguments belong to a class which is a superclass of date)

See, what runtime polymorphism is doing is that if you are trying to invoke a method on an object, it will run the *most specific version* of that method available. It doesn’t matter if you have a manager of static type employee, the most specific dojob() gets called.

Here, that would ideally mean that our equality method gets called, but it ***doesn’t****,*because the function being called, and the function we defined have different signatures.

If say, we created an equals method for the employee class that takes an argument of type employee, then when invoking equality with a manager argument, the signature of “argument is a manager” has **no function** defined for that exact signature, so Java will find the functions of this name (here, equals), look at their signatures and pick the one with the *most specific signature*: Here, that will be boolean equals(employee e){…}.

This is an important lesson: if overloading is occurring, that function which is tailored for most specific class suitable for the static type of the argument is taken. If *overriding* is occurring, that function from the most specific class is unambiguously used.

If we wanted to avoid overloading the OG equals function and create a custom function, then we wouldn’t be able to call it while the find function uses static type Object for o. Only because overriding occurs, the static “object” object is able to invoke the custom equals. To put the pitfall concisely, we overloaded instead of overriding.

To implement the find function properly, we need to define the equality of the date class with the same signature as the Object class, so that equals is properly overridden.

public boolean equals(Object d) {  
 if (d instanceof Date) {  
 Date myd = (Date) d;  
 return ((this.day == myd.day) &&  
 (this.month == myd.month) &&  
 (this.year == myd.year));  
  
 }  
 return (false);  
}

Observe that we need to ensure d is a date type before we proceed, and secondly, we need to create a typecasted object for our purposes. Why? if the object being passed into the function has static type of a superclass, then calling attributes or methods will result in compilation errors.

(Recall, e.doManaging(); throws an error because e is an employee object, and static checking tells that such a method does not exist.)

(Also, note that the object d is declared to be of static type d in the definition, but the actual static type of d when the function is invoked is almost certainly not Object, but some other more specific class.)

Yet another side note: There’s a technical difference between subtyping and inheritance.

“In the object-oriented framework, inheritance is usually presented as a feature that goes hand in hand with subtyping when one organizes abstract datatypes in a hierarchy of classes. However, the two are orthogonal ideas.

* Subtyping refers to compatibility of interfaces. A type **B** is a subtype of **A** if every function that can be invoked on an object of type **A** can also be invoked on an object of type **B**.
* Inheritance refers to reuse of implementations. A type **B** inherits from another type **A** if all functions of **B** can be written in terms of functions of **A**.

However, subtyping and inheritance need not go hand in hand. Consider the data structure deque, a double-ended queue. A deque supports insertion and deletion at both ends, so it has four functions insert-front, delete-front, insert-rear and delete-rear.

If we use just insert-rear and delete-front we get a normal queue. On the other hand, if we use just insert-front and delete-front, we get a stack. In other words, we can implement queues and stacks in terms of deques, so as datatypes, Stack and Queue inherit from Deque. On the other hand, neither Stack nor Queue are subtypes of Deque since they do not support all the functions provided by Deque. In fact, in this case, Deque is a subtype of both Stack and Queue!”

### Summary

1. Classes contain 2 things: variables and methods.

- Variables, by default, belong to objects.

- The static modifier makes a variable a class variable.

- The final modifier makes a variable impossible to change once instantiated.

- The private modifier makes a variable inaccessible outside the class. This means that class code can use private variables of all instances of the class.

- The protected modifier makes a variable accessible within its class and all its subclasses.

- The default behaviour of a variable is that it is accessible only within its package. With the public modifier it becomes openly accessible in any package.

2. Methods

- A method is described both by its name and the types and number of its arguments. This is called the signature of the function. 2 Methods can exist with the same name but different signatures, so they are different entities. This is called overloading.

- The static modifier makes a method callable without an instance. (object)

- The final modifier makes a method impossible to override.

- If unspecified, a method is package level. It’s not possible to call it outside the package. Default (Package), Private, Protected and Public behaviour is same as variables.

3. Constructors

- Constructors are special methods that bear the name of the class. Every constructor calls the default constructor of its parent class, if the constructor in question does not explicitly invoke a super constructor. The keyword this(args) is used to call alternate constructors of the same class within a class. This can be used to chain constructors. The constructor chain leads till Object.

4. Polymorphism and Subclassing

- A variable of static type of class A can be initiated to be an object of any subclass of A.

- Static typing prevents the invocation of any method not available to a class A by an object of static type A, even if it may in actuality be a subclass. To call such a method, typecasting is done.

- If a method is invoked by an object A, then, regardless of the static type, the most specific version of the method available to the object A is used. This is called runtime polymorphism.

- When a method is called, it will execute successfully only if the arguments are of the correct types. It is acceptable for an argument of class A to be replaced by an object of a subclass of A.

- When a method is called, and there are multiple *overloaded* functions with the same number of arguments, fully compatible with this function call, that function which is most specific to the **static types** of the arguments is used.

- **A static method cannot be overridden.** It can, however, be redefined. If an object is used to call a static method (which it should not,) then the one belonging to the static type of the object is used. No polymorphism.

- All classes are subclasses of the class Object.

- runtime/functional polymorphism refers to how a collection of objects each know what they should do although a single common method is called on all of them. The single method is taking many forms.

- Structural polymorphism refers to how a piece of code only expects its arguments to conform to a certain structure, and if it does, then it can act upon the objects, regardless of what else they may be. The single method is acting on many objects, thus, in a way, possessing many forms, akin to overloaded functions.

# Part 3: Interfaces

### 1. Interfaces

Let’s begin with a motivating example. We wish to make a bunch of classes Square, Circle, Rectangle and so on. Clearly, they all share some commonality, so it makes sense to keep all of them as subclasses of a Shape class.

Now, in the declaration of a Shape class, we wish to mention that there exists a function called perimeter, which will allow us to use dynamic dispatch to call all perimeters at once. However, this perimeter function cannot be defined for an unknown shape; It has to just show the signature, and return a garbage value, say, -1. We need to hope and pray that the developers extending your class faithfully implement perimeter, and don’t simply forget about it.

That doesn’t sound so good now, does it?

That’s why we have “abstract methods”; A method that has simply no implementation, and only a signature. Even if there is one abstract method in your class, the class must be declared to be an abstract class. There may be non abstract methods too, but that’s okay. It will simply be impossible to make a subclass without defining all abstract methods concretely.

public abstract class Shape{  
 public abstract double perimeter();  
 public static void sayHello(){  
 System.*out*.println("Hello");  
 }  
}

Because this class itself is now an abstract entity, you **cannot** create objects of this datatype.

However, you can choose this class to be the static datatype of your object.

Shape shapearr[] = new Shape[3];  
int sizearr[] = new int[3];  
shapearr[0] = new Circle(...);  
shapearr[1] = new Square(...);  
shapearr[2] = new Rectangle(...);  
for (i = 0; i < 2; i++) {  
 sizearr[i] = shapearr[i].perimeter();

// each shapearr[i] calls the appropriate method  
}

Now, let’s look at yet another example of the use of abstract classes: sorting. We wish to define a class called Comparable, and all subclasses of this have a method which we define as an abstract up front, called compare, which returns -1 if a<b, 0 if a=b, and 1 if a>b.

A sort can be defined for a list of Comparable objects, and it will work for any subclass. That’s great news. Now, I want to sort my circles. That means circle must extend both Shape **and** Comparable. It must inherit from more than one class.

That’s not something we have seen before. Let’s say class A has a function f. class B has a rival function f, with the exact same signature. What happens if I create class C extending A and B? Which version of f must it keep? Won’t that break the entire idea behind subclassing, that a child can do whatever the parent can?

That’s why Java decided, nope, we will simply not allow multiple inheritance ever, even if there is no conflict. Other languages like C++ allow multiple inheritance if there is no conflict, it’s just something Java decided against.

Now, we have hit a wall. We cannot use Circle extends Shape,Comparable. Sad. What to do? Well, if you recall, Comparable has only one method, and that’s abstract. How can a purely abstract class ever cause a collision of definitions? There’s never even going to be an implementation in an entirely abstract class, so you can extend even 20 of them, and never face a conflict.

That’s why classes that are entirely abstract are given a special name: Interfaces, and a special keyword to replace extends: implements.

class Circle extends Shape implements Comparable{…}

the word implements can be followed by any number of interfaces, but of course, the Circle class has an obligation to concretely implement every single abstract method in each interface.

Life was good, until one day, people started asking Java for implemented functions to be permitted in interfaces. Why? Well, people wrote some code, made it public and called it a day. The next day, they realise, alright, we wish to add an extra function to this interface.

Now, literally all the code that ever used the interface breaks, because you now have to go and add this method to every single class that implements this interface. That’s why Java said, “Okay, I’ll throw you a bone.” And released default methods: methods that need not be implemented in a subclass.

And then they realised they’ll have to worry about the same conflicts they would have run into if they allowed multiple inheritance of classes due to conflicts in these functions. Sigh.

There are actually more reasons why people wanted default functions; they can be used like a placeholder with a garbage return value, like what we tried before abstract. After this, due to more reasons, static functions were also permitted in interfaces.

Now at this point, I’d say Java basically has multiple inheritance. Here’s the policy: If two interfaces have rival functions with the same signature, then the class implementing them both must necessarily create its own version of that function and override them both. If the interface has a rival function to that of the parent class, then the parent class wins.

What even is the difference between abstract classes and interfaces now? For one, interfaces cannot have a constructor, while abstract classes can. They will be a part of the constructor chain of its subclasses. Secondly, an abstract class can have any type of method. An interface can only implement static or “default” methods apart from abstract methods. All variables used in an interface are final and static, while any variable type can be used in a class.

Admittedly, although the difference is clear, it doesn’t quite make sense when to use which one, and how each can be used to their full potential. We will look at 3 examples where interfaces are useful.

### 2. Iron Fists: Confining code within a class

Motivating example: We are making a class to query a railway website.

public class RailwayBooking {  
 private BookingDB railwaydb;  
  
 public int getStatus(int trainno, Date d) {  
// Return number of seats available  
// on train number trainno on date d  
...  
 }  
}

Let’s say we hand off this object to the user end (pretend; that’s not how exactly it works), and they can call getStatus as many times as they like. Bad idea. People will spam the server, do a DDoS Attack, or some other malicious garbage.

To prevent it, first, we check if the user is logged in before we allow them to use this function. Passing username or some such thing into getStatus is one solution, but a bad one that makes for a nonsensical signature. We should make a class to separate out the conern of checking if the user has logged in.

Instead of directly allowing the call of the getStatus method, we create a function that returns an object called query object, which has the get status method. We will check for login, and if and only if the person is logged in, we will give them the QueryObject.

public class RailwayBooking {  
 private BookingDB railwaydb;

public QueryObject login(String u, String p){  
 QueryObject qobj;  
 if (valid\_login(u,p)) {  
 qobj = new QueryObject();  
 return(qobj);  
 }  
 }

private class QueryObject {  
 public int getStatus(int trainno, Date d) {  
 // Return number of seats available  
 // on train number trainno on date d  
 ...  
 }  
 }  
}

If we made QueryObject a public class, then anyone will be able call getstatus again. Bad idea. That’s why the class has to be private, only meant for use within the Railwaybooking class.

This is why private classes are also known as inner classes. They occur only in the inside of other classes. They are basically a part of the parent class, so private variables and methods of the parent class are all accessible in the nested class.

But then, now we have returned this object to the user which is a private datatype. There’s no way the user is going to call methods on that, right?

RailwayBooking x = new RailwayBooking();  
??? qobjforme = x.login();

Notice that we are unable to use a type for the qobj we are getting, because it’s not possible to use that private class here. If we could, it would cease to be private, wouldn’t it?

Okay, so we need a Dummy class, and then we can make the QueryObject class extend that class; Then we can declare qobjforme as an instance of the public Dummy class, right? Well… The static type check will tell: “Dummy has no method called getStatus, so we cannot let you do that”.

Well, then we just need to declare all the methods of the private class within the Dummy class, but as abstract methods, so that things work. Sounds like an interface, doesn’t it? This is one use, to allow access to the public methods of a private class.

public interface QIF{  
 public abstract int getStatus(int trainno, Date d);  
}

public class RailwayBooking {  
 private BookingDB railwaydb;

public QIF login(String u, String p){

QueryObject qobj;  
 if (valid\_login(u,p)) {  
 qobj = new QueryObject();  
 return(qobj);  
 } // This code is incomplete as not all pathways result in a

// correct return statement.  
 }

private class QueryObject implements QIF {  
 public int getStatus(int trainno, Date d){  
 ...  
 }  
 }  
}

Looks good. Works.

But uhh people can now login and then spam you. Or just pass around this QIF object to other people and make an infinite number of queries.

Okay, we clearly need to limit the number of queries permitted, and while we are at it, might as well add a timeout of 20 minutes for the object. This can be accomplished with object variables in the QueryObject class that are checked within the getStatus method. If the limits are exceeded, then nothing is returned. I won’t bother showing the implementation of that here, since the interfacing part is already over.

The above example allowed us to completely encapsulate implementations within a class, and keep it inaccessible to the outside. That’s why I titled it “Iron Fists”. Onto the next example.

### 3. Callbacks

You’ve kept something on the stove, and it will cook for 10 minutes. Once it is done, an alarm is sounded and you switch off the gas and do something. During this time, you do not just stand there and wait for the cooking to get over; you go and chop vegetables during these 10 minutes.

What’s happening is that there is a main thread which triggers a timer for 10 minutes, which then comes back and invokes a function once it is done, and the main thread can proceed without worry during this time.

This is a callback. A function that is invoked asynchronously after some event occurs.

public class Main{  
 public static void main(String[] args) {  
 MyClass m = new MyClass();  
 m.startCooking();  
 }  
}

class MyClass {  
 public void startCooking() {  
 System.*out*.println("Started Cookin!");  
 Timer t = new Timer(this, 3000); // this object created t  
 Thread th = new Thread(t);  
 th.start();  
 }  
  
 public void timerdone() {  
 System.*out*.println("Switched off the stove");  
 }  
}  
  
class Timer implements Runnable {  
  
 // Timer can be invoked in parallel  
 private final MyClass owner;  
 private final int time;  
  
 public Timer(MyClass o, int t) {  
 owner = o; // My creator  
 time = t;  
 }  
  
 public void run() {  
 try {  
 Thread.*sleep*(time);  
 }  
 catch(InterruptedException e) {  
 System.*out*.println("got interrupted!");  
 }  
 owner.timerdone(); // I’m done  
 }  
}

The Main function calls the startCooking function which creates a separate thread, mapped to the “Runnable” object t, of class Timer.

A Thread object is given a runnable object, and the thread will independently execute the run function of the runnable class when the start method of the Thread is invoked.

The timer’s run function sleeps for “time” milliseconds, and then resumes execution. It is necessary to make a try catch, because the sleep function can get interrupted by some other more important thread.

Simple.

Only, this timer is tailored for MyClass. The type of the owner is declared to be MyClass, and that’s what is taken as the input in the constructor. Even if we generalise it and replace these with Object, when you try to do owner.timerdone, you must cast the object into MyClass.

Whatever will we do? Make an interface: TimerOwner.

interface TimerOwner{  
 void timerdone();  
}  
  
class MyClass implements TimerOwner{...}  
  
class Timer implements Runnable {  
  
 // Timer can be invoked in parallel  
 private final TimerOwner owner;  
 private final int time;  
  
 public Timer(TimerOwner o, int t) {  
 owner = o; // My creator  
 time = t;  
 }  
  
 public void run() {...}

}

This use of interfaces is what we started with: A broad group which can be used to identify a common functionality present in a diverse set of class.

The third use of interfaces I wish to show is that of Iterators.

### 4. Iterators

An iterator is used to traverse an iterable object without having to know its internal implementation or access rules. For an array you may use arr[i] and loop through i but if the object is a complex class, moving through this object and pulling out the entities may be a complicated endeavour.

Also, this implementation of iteration should be kept private. There is no reason to allow the iter to be directly called, it’s not meant for such usage.

When you think about it, a for loop and a while loop are identical in capability. The C style for loop is honestly kind of redundant, and that’s why python did away with it. Java has a second version of the for loop, which is closer to what python has.

for (type x: a){  
 do something with x;  
}

This for loop implicitly uses an Iterator in the background. Just as python uses \_\_iter\_\_ and \_\_next\_\_.

Before we try to implement an Iterator using the default Java interfaces, which will let us use this version of the for loop, we will develop our own version of Iterator.

An Iterator needs to have 2 methods: hasNext() and next(), both of which are pretty self explanatory.

Linearlist l = new Linearlist();

Object o;  
Iterator i = l.getIterator();

while(i.hasNext()){  
 o = i.next();  
 ... // do something with o  
}

This is how we would iterate through the Linearlist l. Note that the implementation of the Iterator is fully unique and tailored to suit the class Linearlist, which is why we need to use getIterator.

The internal implementation will be something like:

public interface Iterator{  
 public abstract boolean hasNext();  
 public abstract Object next();  
}

public class Linearlist{  
 private class Iter implements Iterator{  
 private Node position;  
 public Iter(){...} // Constructor  
 public boolean hasNext(){...}  
 public Object next(){...}  
 }

// Export a fresh iterator  
 public Iterator getIterator(){  
 Iter it = new Iter();  
 return(it);  
 }  
}

Observe the similarity to the first example; a private class is used to implement a feature, and the external world uses an interface to engage with it. The public getIterator method dishes out a private object for usage.

This is the basic idea.

Now, let’s look at how to implement the real Java Interfaces.

Firstly, the class you wish to iterate through, must implement “Iterable”. (A use similar to the 2nd example, grouping a large number of classes with a common feature.)

The interface Iterable has exactly 1 abstract function, which makes it a *functional interface.* Other examples of functional interfaces are Runnable and Comparable. (But all of these are poor examples for reasons that will be obvious once we get into functional interfaces) From Java 8, something called a lambda expression can be used to define anonymous instances of functional interfaces, but we will get to this as well later.

The interface Iterable asks only for one function: public Iterator iterator(); Something that can return an Iterator object. The Iterator interface has 2 abstract functions: hasNext and next. Once these have been set up the for each loop will work.

Except, your compiler will probably throw a warning about raw classes. To correct this we must understand how interfaces and classes can be generalised to any type of object.

### Summary

1. Methods

- An unimplemented method can be declared in a class with the keyword abstract. This renders the class incapable of spawning any objects ever. The class must also be given modifier abstract.

- An interface is an unimplemented class with almost entirely abstract functions. Several updates to Java allowed more methods to be permitted in interfaces:

* Default methods are inherited by all who implement the interface. They need not be overridden.
* Static methods are also inherited. They *cannot* be overridden, only redefined.
* Private methods are merely for the purpose of encapsulating code to serve static and default methods. They are inaccessible outside the interface.

- Any non default/static function is automatically abstract in an interface, so the keyword is rarely used before the methods. Similarly, any non private method is public and this cannot be changed.

- Nonsense: private default, default static, private abstract, static abstract, default abstract

2. Inheritance rules

- An interface can **extend** any number of interfaces.

- Any number of interfaces can be implemented by a class. All abstract functions in all interfaces **must**\* be implemented concretely in this class.

\* Caveat 1: If an interface A has an abstract function f, and another interface B has a *static* function f, then an a class C that implements both A and B does not have to provide an implementation for f since it inherits the static function from B.

- However, if instead of static, the method in interface B was default, class C must necessarily provide its own implementation and override the default method. (This can be seen as a consequence of default being created to be an “optional placeholder”.)

\* Caveat 2: If a class C that extends a class B and implements an interface A, inherits a function f from B which A has as abstract, then the implementation need not be done in C. However, this function in B must be public to avoid the implementation, as the interface specifically requests for a public function.

3. Conflict Resolution

- if a class C inherits default function f from both interfaces A and B, then C must implement f.

- if f were static, then there is no such requirement, since an interface’s static method can only be invoked with if\_name.f() and not with an object.

- if a class C inherits default function f from interface A and f of same signature from class B, then C happily uses the class B one, since default functions are made to be overridden.

- if f were static, then there is no issue. Object.static\_method (which is bad practice,) will call the class version, since interface static methods***cannot*** be invoked by instances.

4. Common uses of Interfaces

- Encapsulating the public methods of a private class, so that external code can interface with it.

- Acting as an umbrella data type that identifies a class to possess a particular functionality. This is key to implementing Structural Polymorphism.

(“Structural polymorphism refers to how a piece of code only expects its arguments to conform to a certain structure, and if it does, then it can act upon the objects, regardless of what else they may be.”)

- An interface with exactly one abstract function embodies exactly one functionality, so it is called a functional interface. “Anonymous” classes (similar to anonymous functions in JS) implementing functional interfaces can be created elegantly using lambda expressions.

- Notable examples: Iterator and Iterable interfaces; The Runnable interface and Thread class;

Addendum: If a class can be implemented internally in multiple ways, and each one has its own pros and cons, one would want multiple classes to choose between these implementations. However, this means that if one wants to swap implementations they must find every mention of this class and carefully change it. Instead, a common interface with all the expected properties can be defined, and that can be used everywhere for object static types or function return types, and then the instantiation alone needs to be changed to swap implementations.

In this sense, an interface is allowing you to interface with an abstract concept and adding a level of “indirection”. You can say that it will be implemented by someone, and write code on top of it. This example not only shows the broader picture of the first point I mentioned, but makes the keywords used a lot more intuitive.

# Part 4: Generics

Structural polymorphism refers to how a piece of code only expects its arguments to conform to a certain structure, and if it does, then it can act upon the objects, regardless of what else they may be.

We attempted to implement this using interfaces, and for the most part, we were successful.

Consider a simple function: Copying an array of objects.

public static void arraycopy (Object[] src, Object[] tgt){  
  
 int i,limit;  
 limit = Math.*min*(src.length,tgt.length);  
 for (i = 0; i < limit; i++){  
 tgt[i] = src[i];  
 }  
}

Date[] datearr = new Date[10];  
Employee[] emparr = new Employee[10];  
arraycopy(datearr,emparr); // Run-time error

Observe that both the emparr and datearr are subclasses of Object, so they get accepted into the function. Then, when the actual copying is done, a type error is thrown.

A *type* error, of all things, is supposed to be caught at compile time, **not** runtime.

We need both src and tgt to be of the same type, right? Well, more generally, src can be a subclass of the type of tgt, and things will work out.

Consider this example of the implementation of a LinkedList:

public class LinkedList{  
 private int size;  
 private Node first;  
 public Object head(){ ... }  
 public void insert(Object newdata){...}  
 private class Node {...}  
}

LinkedList list = new LinkedList();  
LinkedList list = new LinkedList();  
Ticket t1,t2;  
t1 = new Ticket();  
list.insert(t1);  
t2 = (Ticket)(list.head());

Date d = new Date();  
list.insert(d);

First, observe that because we wanted to store a general element, we declared its type to be Object inside the LinkedList. As a consequence, we completely lose information about the type of the object when we retrieve it. Further, It is possible to insert elements of a random class, here, Date, and pollute the homogeneity of the List.

This is a disaster on many levels, and this directly leads us to generics.

public static <S extends T,T> void arraycopy (S[] src, T[] tgt){  
  
 int i,limit;  
 limit = Math.*min*(src.length,tgt.length);  
 for (i = 0; i < limit; i++){  
 tgt[i] = src[i];  
 }  
}

We begin with those angle brackets, a bit like how in mathematics, we say “for all x,y such that…”. That says “For all classes S and T, where S extends T, the following function holds.”

Now, if we wish, we can use the class type S and T within our function code block as well.

Here’s a polymorphic reverse:

public static <T> void reverse (T[] objarr){  
 T tempobj;  
 int n = objarr.length;  
 for (int i = 0; i < n/2; i++){  
 tempobj = objarr[i];  
 objarr[i] = objarr[(n-1)-i];  
 objarr[(n-1)-i] = tempobj;  
 }  
}

And here’s a polymorphic find: (where searching for a target of incompatible type is now a compile time error)

public static <T> int find(T[] objarr, T tgt) {  
 for (int i = 0; i < objarr.length; ++i) {  
 if (tgt.equals(objarr[i])) {  
 return i;  
 }  
 }  
 return -1;  
}

Note that the <> types always come right before the return type.

Let’s try to implement the LinkedList we started off with, having acquired some knowledge of Java Generics.

public class LinkedList{  
 private int size;  
 private Node first;  
 public Object head(){ ... }  
 public void insert(Object newdata){...}  
 private class Node {...}  
}

The argument of insert must be a type T.

The return type of head must be the same type T.

Node must internally store only that same T.

Clearly, the generic class type we want cannot just be prefixed before a function like before. It’s for the entire class.

public class LinkedList<T>{

private int size;  
 private Node first;

public T head(){  
 T returnval;  
 ...  
 return(returnval);}  
 public void insert(T newdata){...}  
 private class Node {  
 private T data;  
 private Node next;  
 ...  
 }  
}

This class definition is read as “a LinkedList of type T”.

Note: When a class is defined with a parameter class T, and a method within the class separately declares a “for every T”, the second T hides, or “shadows” the original T, potentially leading to chaos. Use a different variable name if the methods need to use a new parameter, and use the same T everywhere if they are always the same. Shadowing is **bad**, please don’t do it.

Now, this is the actual Iterator interface from the Java source code.

public interface Iterator<E> {

boolean hasNext();  
  
 E next();

default void remove() {  
 throw new UnsupportedOperationException("remove");  
 }

default void forEachRemaining(Consumer<? super E> action) {  
 Objects.*requireNonNull*(action);  
 while (hasNext())  
 action.accept(next());  
 }  
}

You can see that there are two default functions we always got away without implementing. When we implement this interface, we cannot forget about the <E>, and if we do, that’s what’s a raw class. It’s a miracle it only throws a warning and not a compilation error.

Here’s the Iterable interface:

public interface Iterable<T> {Iterator<T> iterator();

...

}

(I’ve omitted 2 default functions.)

Let me demonstrate how we can implement an Iterator, by creating an useless class that’s basically an array. (Explanation after the code)

class Main {  
 public static void main(String[] args) {  
 Sequence<Integer> sObj = new Sequence<>(5);  
 Scanner sc = new Scanner(System.*in*);  
 for (int i = 0; i < 5; i++) {  
 sObj.addToSeq(sc.nextInt());  
 }  
 for (Object x : sObj) {  
 System.*out*.print(x + ", ");  
 }  
 }  
}  
  
class Sequence<T> implements Iterable<T>{  
 private T[] iArr;  
 private final int size;  
 private int i=0;  
  
 public Sequence(int size) {  
 this.size = size;  
 iArr = new T[size]; // Throws compilation error; We’ll worry later.  
 }  
  
 public void addToSeq(T elem) {  
 if (i < size) {  
 iArr[i] = elem;  
 i++;  
 } else {  
 System.*out*.println("Sequence is full.");  
 }  
 }  
  
 public Iterator<T> iterator() {  
 return new SeqIterator();  
 }  
  
 private class SeqIterator implements Iterator<T>{  
 int indx;  
 public SeqIterator(){  
 indx = 0;  
 }  
 @Override  
 public boolean hasNext() {  
 return indx < size-1;  
 }  
  
 @Override  
 public T next() {  
 if (hasNext()) {  
 T nextElem = iArr[indx];  
 indx++;  
 return nextElem;  
 }  
 return null;  
 }  
 }  
}

That’s probably our most complex block of code yet. Ignore the main function for now.

A Sequence of type T, is an implementation of Iterable of type T. Those brackets and type are part of the class **name**. We are defining a parametrised type, Sequence, and the parameter is T. Now that T has been defined, “implements Iterable<T>” can be done. Iterable cannot exist in isolation, you must give a prereferenced class to even make the declaration.

Sequence is implemented internally using iArr (internal Array) of type T. The size and the index upto which elements have been inserted, i, are both private, and must not be tampered with.

The Constructor initiates, the addtoSeq function adds, and the obligatory iterator function returns an Iterator<T> object. Since the iterator itself is private, the actual object we return is of the private class SeqIterator, an implementation of Iterator<T>.

The “next” function of SeqIterator returns only objects of type T as specified by the interface Iterator. The @Override is a sanity check for the compiler. It will throw a warning if there is nothing to override or if there is some other issue. It’s good practice to put that tag above functions you override.

All this is quite straightforward. The main function instantiates an object of this generic class. First, we must specify the type of class in the angular brackets to lock down what type is acceptable for all methods in the class.

The Constructor call doesn’t need the type within the brackets though, since the class type is already declared.

With all the glorious iterators set up, the for each loop works seamlessly. Except for that one error we get when we compile the code.

How Java implements generic classes is by using something called **Type Erasure.**

Each time we create a LinkedList<Integer> or LinkedList<String>, you might imagine we need to create an entire new class definition for it. That’s just wasteful, though. So, Java just takes all these parametrised types and throws them into the trash can.

It replaces **every single parameter class** with **Object**. Isn’t that **BAD?**

Well, why did we need generics? Couldn’t we just use Object everywhere?

1. The relationships between the types of generic arguments could not be enforced by the compiler since all types are compatible with object.

2. A class cannot maintain internal consistency about what type of object it is storing, and when it is taken out of the class, it needs to be typecast manually since it is stored only as an Object.

Java solves this by allowing you to make generic classes, and then enforcing those rules at compile time, and once all these rules have **been checked**, Java just uses code with Object, because anything that could go wrong, has already been caught.

arrayCopy will never be called with two incompatible typed arrays, because type checking would have caught that, searching for a target of incompatible type with the polymorphic find function is a compile time error and so on.

But how will LinkedList work?

public class LinkedList<T>{

private int size;  
 private Node first;

public T head(){}  
 public void insert(T newdata){...}  
 private class Node {}  
}

LinkedList<Ticket> list = new LinkedList<>();  
Ticket t1,t2;  
t1 = new Ticket();  
list.insert(t1);  
t2 = list.head();

Becomes

public class LinkedList<Object>{

private int size;  
 private Node first;

public Object head(){}  
 public void insert(Object newdata){...}  
 private class Node {}  
}

LinkedList<Ticket> list = new LinkedList<>();  
Ticket t1,t2;  
t1 = new Ticket();  
list.insert(t1);  
t2 = (Ticket) list.head();

Return values of type T, like in the function “head”, are all automatically typecasted by Java into the corresponding type, so that no code breaks.

Any argument types of T are just used with Object, and no harm will come of that.

And no method or constructor can ever be invoked with T, because all we know after type erasure is that T is some subclass of Object.

This is part of the reason why new T[size] is not allowed. The compiler cannot instantiate a class it does not know anything about. What it can do, is instantiate Object[size].

Although the compiler will not throw an error, it will through a warning for the following line which can replace our line in the older code, and make it actually work:

iArr = (T[]) new Object[size]; // Gives warning

It’s an ugly workaround, and it’s just as bad as the previous line, honestly. This is an unchecked typecast, that might throw a runtime error.

When I said that every place T is present is replaced with Object, I was slightly incorrect. If T is declared *bound* to a certain class, like <T extends Number> then T will instead be replaced with Number rather than Object within the code, a sensible decision, which allows *some* methods and constructors to be called.

Also, it is because T must be replaced with Object, that LinkedList<int> and such is not possible. The 8 basic scalar types of Java are **not** compatible with Object.

This is why all 8 of them have Wrapper classes, Integer, Double, Float etc. And all the 6 number scalar types are subclasses of type Number.

This suggests that in order to use Integers when they are passed into LinkedLists, or any parametrised class, the scalar value needs to wrapped, and then when we pull out the data, we need to “unbox” the integer.

int x = 5;  
Integer myx = new Integer(x);  
int y = myx.intValue();

This is dumb.

Which is why Java does it automatically in the background for you. The above code actually throws a deprecated constructed error, because Java now “Autoboxes” for you.

int x = 5;  
Integer myx = x;  
int y = myx;

int and Integer can be used INTerchangably.

Before we conclude this section, we must look at how Subclassing interacts with Generics.

In OOP literature, there exists a concept called covariance. If a class B is a subclass of A, then B in any context, is a subclass of A in that context. For instance, B[ ] is a subclass of A[ ].

An array of As can be replaced by an array of Bs.

Java is covariant, and this is actually not a good thing, sometimes.

EvenNumber[] x = new EvenNumber[10];  
NaturalNumber[] y = x;

Because EvenNumber is a subclass of NaturalNumber, y can be used in place of x, much like how Employee y can be used in place of Manager x.

The thing is, now, when you try to add a natural number to y, because it is internally an array of the subclass, it will throw an error.

Basically, Covariance can be a dumb idea, so they decided they won’t do it for Generics.

Now, LinkedList<EvenNumber> is **not** a subclass of LinkedList<NaturalNumber>. This means that if a method argument called for LinkedList<NaturalNumber>, it will **not accept a** LinkedList<EvenNumber>, even if in theory, you’d imagine they behave covariantly.

Thus, if you wanted to allow for a generic LinkedList, you cannot define a method for LinkedList<Object>, that will permit **only Object.** You must use a generic type T.

public static <T> void printlist(LinkedList<T> l){}

­If the function’s implementation does not use the type T anywhere, you can simply do:

public static void printlist(LinkedList<?> l){}

The ? cannot be used anywhere concretely, it’s like a nameless variable, just used to avoid the usage of an extra name <? extends Shape> and such are also acceptable.

### Summary

- Methods and classes defined purely with Object class arguments in order to be general potentially result in runtime errors. Relationships between arguments cannot be enforced, and objects of only class Object can be returned, causing irritating typecasting which could also cause errors at runtime.

- To solve both these problems Generic types were introduced.

- A parameter type can be declared along with a class, which makes the class a “parametrised type”, that requires a parameter in its declaration.

- A parameter type can be declared along with a method, which allows a generic return type and input pattern along with convenient types in the function body. If the parameter shadows a class parameter, the method parameter is used, and the class parameter is forgotten.

- Internally, Generics are implemented by Type Erasure. This means that the class and method definitions’ parameters are all replaced with the type Object, (or the closest bound class,) and any returned objects are auto typecast. Due to erasure, parameter classes cannot call methods not present in their closest bound class, including constructors.

- Further, functions cannot be overloaded if their signature differs only in generics. printAll(LinkedList<Date>) and printAll(LinkedList<Money>) is not allowed, since there is only LinkledList<Object> at runtime, **nothing** else.

- Scalar datatypes are allowed to exist despite Java being an OOP language, but due to incompatibility with the rest of Java, wrapper class with Capitalisation are used in their place. These are autoboxed by Java for convenience.

# Part 5: Dealing with things at Runtime

You might imagine nothing interesting happens at runtime in Java with the level of compile checks it has, but runtime is the most interesting time. Some things need to be decided based on what happens when a line of code is executed. Firstly, the most simple example of this is Reflection.

### Reflection

It is the ability of code to introspect at runtime and choose to make changes in the control flow based on that.

- Can verify the dynamic type of an object by introspection at runtime: (e instanceof Manager)

- Can find out the class of an object at runtime with o.getClass(). getClass() is a method defined in Object.

Unlike python’s type, this getClass() does not return just a simple string; it has a much richer structure.

getClass() returns a “Class” object. (more strictly, of type Class<T>, since T is part of the name. T represents the class modelled. For a String Object, getClass() will return an object of type Class<String> and so on.)

For each class in existence, Java internally makes a Class Object.

Dummy obj = new Dummy();  
Class<?> c = obj.getClass();  
Object o = c.newInstance();

This is a very bad example for 2 reasons, one is that newInstance is a depreciated method, and two, the code throws an unhandled exception error, but I will get to that a little later.

A class is defined by

1. Variables, or Fields

2. Constructors which possess arguments.

3. Methods, which possess arguments and a return type.

All 3 also possess modifiers like final, static, public etc.

All this information is in the Class Object. To better capture this substructure, Constructor, Method and Field Classes exist. They have their own methods and stuff.

Object o = c.newInstance();  
Field f = c.getField("i");  
Method m = c.getMethod("Hi");  
Constructor<?> con = c.getConstructor();

All these lines throw an unhandled exception error. Why? What if the requested method or field does not exist? What is the code supposed to do then? Just Break? That’s dumb.

Therefore, you must catch any known, obvious, possible exception and deal with it. Anyway, Constructor alone takes arguments of parameter types, while the others take a string argument.

Field[] fs = c.getFields();  
Method[] ms = c.getMethods();  
Constructor<?>[] cons = c.getConstructors();

This doesn’t throw an error, and is convenient tool.

m.invoke(args) invokes the method.

con.getParameterTypes does exactly what it says on the box. Returns an array of type Class<?>.

f.get(obj) returns the value of field f in object obj. This can throw an illegal access exception if the field is private.

However, there are some methods in the reflection standard library java.lang.reflect which access private things too. This means that the privacy in Java in not absolute, and is invadable with the right tools. This is seen as acceptable because the meta analysis that ‘reflect’ offers is typically used only by debuggers running scripts to run tests on code.

However, for people with legitimate concerns, the only escape is sandboxing of the java environment and preventing people from running whatever they want.

Anyway, let us move onto some more important runtime mechanisms: Exception handling, Error Logging, and Assertion checks.

### Errors and Logs

If there is a file name being written as a string, the file may not exist in that path, or we may not have permissions for that file. There is no guarantee that that line of code will run.

If a command requires an internet connection, or it relies on some other device, such a printer, that device can fail.

If a command is writing to disk, the disk can become full.

Apart from that, there are simpler code errors, accessing a key not in a dictionary or an index not in an array, dividing by zero etc.

Python’s approach is to just not worry about it when coding, and if something goes wrong, the code will terminate. It’s upto the programmer’s discipline to put try except statements wherever it may seem necessary.

Java enforces things a little more rigorously. If it is possible for something bad to occur, and you know it, you *must* deal with it.

As always, there is a class for Exceptions, and all exceptions descend from class Throwable. Throwable has 2 children, Error and Exception. Error handles things that go wrong within the JVM, and is not the programmer’s fault, in some sense.

Exceptions are supposed to have been checked for and caught by the programmer. Exceptions are broadly of two types, RunTimeExceptions, thrown by Java, and checked exceptions, thrown by us, when we want to signal that something anomalous has occurred.

When a function throws an exception, it terminates, and goes to the function that invoked it. If that function does not catch this exception, that function also terminates, and the same exception keeps ascending the functions. Eventually, if nobody has caught it, then the starting node (main) will fail, and the program itself terminates.

public static void myfunc() throws ServerException {  
  
 try{  
 some database access  
 }  
 catch (SQLDataException e){  
 String emsg = " More details on why" + e.getMessage();  
 ServerException newe = new ServerException(emsg);  
 newe.initCause(e);  
 throw e;  
 }  
 catch (...){  
 ...  
 };  
 finally{  
 terminate db connection  
 }  
  
 }

There are 3 things I intend to show with this example: first is that errors have an internal structure. The cause for an error to have been initiated can itself be an error. At the final stage, these errors can be inspected deeply to understand the root cause. If we threw a ServerException without initiating the cause, the SQL exception would’ve been fully lost, except for the message we retrieve when we make the error message string.

The second thing is that any exceptions a function can possibly throw has to be listed *after* the function signature. This is obviously, not a part of the signature itself.

The third is that sometimes, exceptions are thrown within the catch blocks, and even so, regardless of whether it is a checkedexception, or an unexpected RunTimeException, we may want to run some code before we exit this method. Here, it is to terminate the DB connection cleanly. Such code is put in the **finally** block, which is executed no matter what happens in the catch blocks.

One additional important thing about exceptions not illustrated here is that the catch blocks are tried from top to bottom. Since exceptions themselves are classes, some are subclasses of others. In such a case, a more specific exception must be caught earlier, so as to take appropriate corrective action.

public static void myfunc(int x) throws IllegalArgumentException{  
 if(x<0){  
 throw new IllegalArgumentException();  
 }

Perhaps, we wish for only positive numbers to be in a function. Ideally, when our software reaches the final stage, we will never invoke this function with a negative argument. This kind of code is really meant only for a debug stage, when we want to be alert to any mistakes made in the construction of the stack.

Since keeping this kind of debug code around is highly inefficient, Java uses assertions.

public static void myfunc(int x){  
 assert x>0;  
 }

This will throw an AssertionError, IF you run the java file from the command line, with the option –ea or equivalently, -enableassertions. You might need to do a little bit of config jujutsu to get assertions to work on your IDE.

Assert lines simply do not execute if the –ea option has not been provided.

Note: assert statements are ***meant*** to flag **fatal** exceptions. Do **not** catch them with try in parent functions; If you want to flag something, and take corrective action, use exceptions, not assertions.

Actually, maybe you want to flag something, but you don’t want to throw an exception. Then what you need to do is make a log. Apart from stating concerning matters, log messages can simply provide information and can act as a way to look back on what happened when the code was running in the past.

Logs are of 7 levels, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST. By default, the top 3 alone are activated. This hierarchy, and the ability to switch on and off logs at various levels without having to edit code is what makes logging far superior to print statements.

Logging is done with the java.util.logging.Logger class (very distinct from the Logger interface in the System Package which we don’t care about) which has a very convenient getGlobal method which returns *the* global Logger Object, which is enough for most purposes.

Any level specified above has a method by that same name to log at that level:

Logger.getGlobal().info(“message goes here”)

To turn on logs until a particular level, the setLevel method has to be used, with an argument of Level.[levelnamehere]:

Logger.getGlobal().setLevel(Level.ALL)

ALL and NONE are two special levels for convenience.

If you do not wish to use the global logger, you can create a new Logger object and give it some name. The cool thing about named loggers is that they are hierarchical. Typically these names are dot separated, and with each word, more specificity is obtained.

For instance, I may have 3 loggers:

math.sign

math.sign.poslogs

math.sign.neglogs

Logger z = Logger.*getLogger*("math.sign");  
Logger y1 = Logger.*getLogger*("math.sign.neglogs");  
Logger y2 = Logger.*getLogger*("math.sign.poslogs");  
  
z.info("Sign logger");  
y1.finer("Pos logger");  
y2.finer("Neg logger");

What do you think happens?

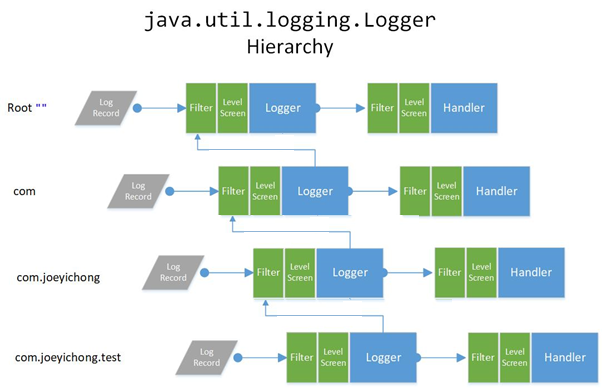
As I said, by default only uptil INFO is logged, so only the Sign logger prints something.

If I do

y1.setLevel(Level.*FINER*);  
y2.setLevel(Level.*FINER*);

Well, surprisingly, nothing changes. Only the info is logged.

That’s because a logger just puts things into a log file and calls it a day. Displaying things on screen, that is, redirecting a copy to stdout, is not a logger’s job; It’s the handler’s.



Let’s take this one step at a time. Look at one row. The execution of the log statement is fed into the level filter of that logger. If it makes it through, it is forwarded to its handler, which has its own level filter, after which it prints the message.

Now, each logger also sends a copy of its log to its parent logger.

In the first version of the code I had written, y1, y2 and z levels had not even been set, so they just didn’t have a level filter. Since y1 and y2 were not given a dedicated handler, they were not logging anything by themselves. y2 forwarded its message to y1, but that didn’t have a handler either, and same case for z. All 3 of them actually sent the message to the root logger, with the name of “” (empty string), and the root handler handled all the logs.

Thus, to make the logs display,

Logger.*getLogger*("").getHandlers()[0].setLevel(Level.*FINER*);

Well… that doesn’t do the trick either. That’s because the Log Level of the global logger has not been changed.

Once that is changed, it does the trick. What do you think the following code outputs?

Logger z = Logger.*getLogger*("math.sign");  
  
ConsoleHandler h = new ConsoleHandler();  
h.setLevel(Level.*FINER*);  
z.addHandler(h);  
z.setLevel(Level.*FINER*);  
  
Logger y1 = Logger.*getLogger*("math.sign.neglogs");  
Logger y2 = Logger.*getLogger*("math.sign.poslogs");  
  
z.info("Sign logger");  
y1.finer("Pos logger");  
y2.finer("Neg logger");

y2 → y1 → z → z handler, logs everything upto FINER

z → “”→ root handler, logs everything upto INFO

thus, y1 and y2 get logged once, but z.info gets logged twice, by both the loggers.

z.setLevel(Level.*WARNING*);

z.info("Sign logger");  
y1.info("Pos logger");  
y2.info("Neg logger");

This will result in NO logs. The z blocks all 3 log statements, so none of them make it to the root logger.

Side note: Packages have a hierarchical structure similar to the Loggers. Not that they involve any such cascading, just that they also have this top down naming. It is common to illustrate it with a website name.

Typically, websites are named like learnjava.ac.in whereas in java, it would be in.ac.learnjava, where we descend down to specifics. The package names also correspond with how they are stored in folders, which makes it even more intuitive, and points out how strange website names actually are from a programming point of view.

Tbh IP addresses do descend that way, not in the order of the literal sitename, and it’s more intuitive from a recognition pov to start with a person’s name rather than their planet or species.

### Summary

- Reflection is the ability of code to introspect at runtime and choose to make changes in the control flow based on that.

- myobj.getClass() returns an object of type Class. It has substructure comprised of Field, Method and Constructor<> objects. Some notable methods are:

* + myclassobj.getMethods()
  + myclassobj.getFields()
  + myclassobj.getConstructors()
  + mymethod.invoke(args)
  + myfield.get(myobj)
  + myconstructor.getParameterTypes()

- Exceptions are major errors in the code that can potentially be mitigated and accounted for. In case of a crash, they act as debug info with rich structure.

- All Exceptions extend the Throwable **class**. Notable fields of this class include cause and detailmessage, both of which are **private**.

- cause is a Throwable object, allowing a chain of errors to be analysed. This can be set using the initCause method, and accessed using the getCause method.

* + detailmessage is necessarily filled when a Throwable object is constructed. Common constructors include Throwable(message), (cause) and (message, cause). It is accessed with getMessage().

- An Exception invoked by the programmer is known as a checked exception, as opposed to RunTimeExceptions which just happen. If any exception is deliberately thrown in a method, that method’s header line must end with a list of all of them.

modifers returntype f(args) throws E1,E2...{}

- Exceptions can be caught with a try block, and they can be handled one at a time with catch blocks. These are checked top down, which must be kept in mind when dealing with Exceptions which are subclasses of other Exceptions. The equivalent of the bare except of python is catch(Exception e). An identifier for the exception must always be provided, unlike python.

- Assertions are used to flag fatal errors during testing phases. Assert lines do not execute unless a –ea flag is used when the file is executed from the CLI.

- Logs are made by Loggers. If configured to do so, a logger simply writes a log to file (unconfirmed). All loggers except root forward the log message to their parent logger and their own handler, if it exists.

- A log statement will not cause a log if it does not clear the filter of the logger. It will not be printed to stdout if it doesn’t clear the handler’s filter.

- If a logger’s level is set to none, then it effectively has the same level as its first parent with a level, since it will be the first one to intercept the message. A handler can receive the message before this as well.

# Part 6: Collections & Maps

In the earliest days of java, java had all kinds of datatypes, vectors, maps etc, but in an unorganised fashion, akin to python. With updates, Java decided to organise these data types into a logical hierarchy so that general methods could be defined.

The Collection Interface defines any data structure consisting of a bunch of elements which can be iterated over. This does not contain Key Value pair structures, which fall under the interface Map.

public interface Collection<E> extends Iterable<E> {int size();  
  
boolean isEmpty();  
  
boolean contains(Object o);  
Iterator<E> iterator();  
  
Object[] toArray();  
  
<T> T[] toArray(T[] a);  
default <T> T[] toArray(IntFunction<T[]> generator) {  
 return toArray(generator.apply(0));  
 }  
boolean add(E e);  
  
boolean remove(Object o);  
  
boolean containsAll(Collection<?> c);  
  
boolean addAll(Collection<? extends E> c);

boolean removeAll(Collection<?> c);  
  
default boolean removeIf(Predicate<? super E> filter) {  
 Objects.*requireNonNull*(filter);  
 boolean removed = false;  
 final Iterator<E> each = iterator();  
 while (each.hasNext()) {  
 if (filter.test(each.next())) {  
 each.remove();  
 removed = true;  
 }  
 }  
 return removed;  
 }  
  
boolean retainAll(Collection<?> c);  
  
void clear();  
  
boolean equals(Object o);  
  
int hashCode();  
  
@Override  
 default Spliterator<E> spliterator() {  
 return Spliterators.*spliterator*(this, 0);  
 }  
default Stream<E> stream() {  
 return StreamSupport.*stream*(spliterator(), false);  
 }  
default Stream<E> parallelStream() {  
 return StreamSupport.*stream*(spliterator(), true);  
 }  
}

Let’s first look at the common features of all Collections, and then at the specifics of each of the data types we know and love.

The key feature of a collection, is the fact that it is iterable. As we’ve already seen, the iterator() method returns the iterator, and we know the iterator functions iter.hasNext() and iter.next().

The *iterator* also has a method remove(), which deletes the element that was just accessed with next. For example, to delete the first element, you’d have to do i.next() i.remove(), not just remove().

It’d sure be convenient if there was a remove function on the *collection,* which goes and deletes the given object. This is left as an abstract method, as is the equals function required to make it work.

Taking it one step farther, removeAll deletes all the objects present in another collection. addAll() extends the object by adding all the elements in the argument collection. containsAll returns true if this collection contains all of the elements in the specified collection.

Note that even add and remove have Boolean return values. Add returns false only in some rare cases, like a set already having an element, and remove returns false when the thing attempted to remove doesn’t exist in the collection.

Overall, Collection has a full zoo of abstract functions. Most of them are not strictly needed for defining a custom collection. Ideally, they’d just be default methods, but sadly, the collection grouping was done before default methods existed in Java.

Thus, at the time, a separate abstract class was created, one that implemented this Collection Interface with a bunch of ready to use, general functions. This class is called AbstractCollection, and extending it is still the easiest way to get started on making a custom Collection.

This one actually has a lot of sensible declarations and only a few abstract methods, so the code is quite long. I’m not going to paste it in here.

Let’s move on to the major Collections.

### 1. List<E>

public interface List<E> extends Collection<E>

At first glance, it will seem that the List interface has a bunch of redundant abstract functions which were already declared in the parent interface. However, these functions now work with *indices.*

There are two ways to access elements: sequentially or randomly. Random Access is what is available in a normal list, where you can access any element at will in constant time.

When making methods that act upon objects of a class that implements List, the method might not be efficient if only sequential or random access is used.

for (int i = 0; i < list.size(); i++)  
// do something with list.get(i);

If list.get(i) was implemented with sequential access, this function will be ***abysmally slow.*** Why? Because every iteration, the get method will start at the beginning of the list and work it’s way back to index i. O(n2) complexity for a simple loop.

Thus, it is important that we make a distinction between SequentialAccess and RandomAccess.

public interface RandomAccess {  
}

That’s literally it, an empty interface. That’s an actual interface in the java.util library. This is a labelling interface that allows us to check if an object isinstance of RandomAccess and if not, run a code suited to sequential access.

Just as there is AbstractCollection, there is AbstractList.

public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {

Further,

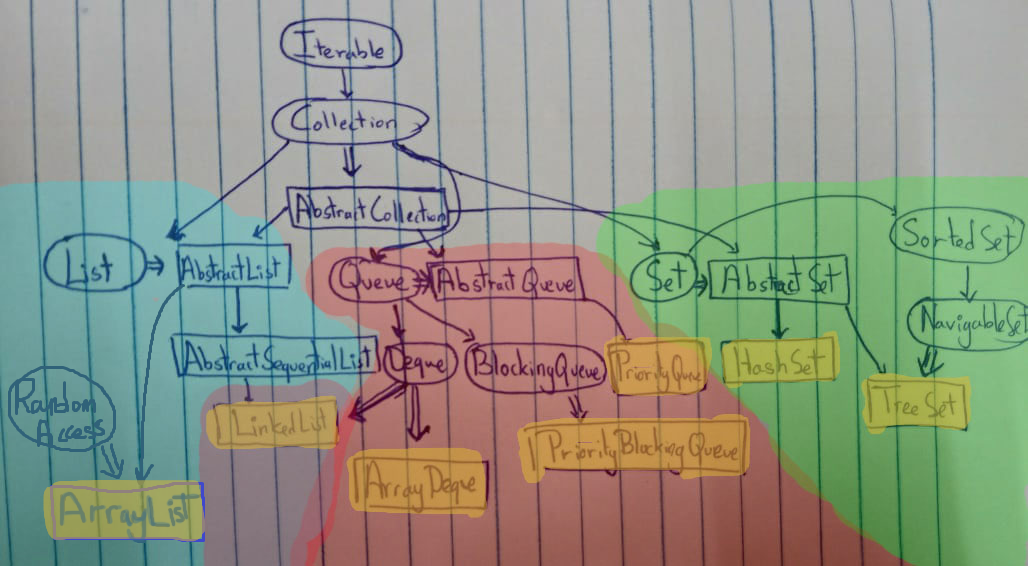
public abstract class AbstractSequentialList<E> extends AbstractList<E>

and

public class LinkedList<E> extends AbstractSequentialList<E>  
 implements List<E>, Deque<E>, Cloneable, java.io.Serializable

and

public class ArrayList<E> extends AbstractList<E>  
 implements List<E>, RandomAccess, Cloneable, java.io.Serializable



I’ve coloured the big 3 extensions of the collections interface RGB, and given gold to actual concrete classes that can make objects. Ellipses are interfaces, in case it wasn’t obvious.

I’ll keep coming back to this, because otherwise it’s impossible to see how all these interrelated species connect.

Lists have a special iterator called ListIterator, which can be used in both directions: It has previous() and hasPrevious methods as well.

ListIterator<E> listIterator();

public interface ListIterator<E> extends Iterator<E> {  
 */\*\*  
 \** ***@return*** *{****@code*** *true} if the list iterator has more elements when  
 \* traversing the list in the forward direction  
 \*/* boolean hasNext();  
  
 */\*\*  
 \* Returns the next element in the list and advances the cursor position.  
 \** ***@return*** *the next element in the list  
 \** ***@throws*** *NoSuchElementException if the iteration has no next element  
 \*/* E next();  
  
 */\*\*  
 \* Returns {****@code*** *true} if this list iterator has more elements when  
 \* traversing the list in the reverse direction. (In other words,  
 \* returns {****@code*** *true} if {****@link*** *#previous} would return an element  
 \* rather than throwing an exception.)  
 \*/* boolean hasPrevious();  
  
 */\*\*  
 \* Returns the previous element in the list and moves the cursor  
 \* position backwards. This method may be called repeatedly to  
 \* iterate through the list backwards, or intermixed with calls to  
 \* {****@link*** *#next} to go back and forth. (Note that alternating calls  
 \* to {****@code*** *next} and {****@code*** *previous} will return the same  
 \* element repeatedly.)  
 \*/* E previous();  
  
 */\*\*  
 \** ***@return*** *the index of the element that would be returned by a  
 \* subsequent call to {****@code*** *next}, or list size if the list  
 \* iterator is at the end of the list  
 \*/* int nextIndex();  
  
 */\*\*  
 \** ***@return*** *the index of the element that would be returned by a  
 \* subsequent call to {****@code*** *previous}, or -1 if the list  
 \* iterator is at the beginning of the list  
 \*/* int previousIndex();

*/\*\*  
 \* Removes from the list the last element that was returned by {****@link*** *\* #next} or {****@link*** *#previous} (optional operation). This call can  
 \* only be made once per call to {****@code*** *next} or {****@code*** *previous}.  
 \* It can be made only if {****@link*** *#add} has not been  
 \* called after the last call to {****@code*** *next} or {****@code*** *previous}.  
 \*  
 \** ***@throws*** *UnsupportedOperationException if the {****@code*** *remove}  
 \* operation is not supported by this list iterator  
 \** ***@throws*** *IllegalStateException if neither {****@code*** *next} nor  
 \* {****@code*** *previous} have been called, or {****@code*** *remove} or  
 \* {****@code*** *add} have been called after the last call to  
 \* {****@code*** *next} or {****@code*** *previous}  
 \*/* void remove();

*/\*\*  
 \* Inserts the specified element into the list (optional operation).  
 \* The element is inserted immediately before the element that  
 \* would be returned by {****@link*** *#next}, if any, and after the element  
 \* that would be returned by {****@link*** *#previous}, if any. (If the  
 \* list contains no elements, the new element becomes the sole element  
 \* on the list.) The new element is inserted before the implicit  
 \* cursor: a subsequent call to {****@code*** *next} would be unaffected, and a  
 \* subsequent call to {****@code*** *previous} would return the new element.  
 \* (This call increases by one the value that would be returned by a  
 \* call to {****@code*** *nextIndex} or {****@code*** *previousIndex}.)  
 \*  
 \** ***@param*** *e the element to insert  
 \** ***@throws*** *UnsupportedOperationException if the {****@code*** *add} method is  
 \* not supported by this list iterator  
 \** ***@throws*** *ClassCastException if the class of the specified element  
 \* prevents it from being added to this list  
 \** ***@throws*** *IllegalArgumentException if some aspect of this element  
 \* prevents it from being added to this list  
 \*/* void add(E e);

void set(E e);

}  
}

It’s a shame Java docstrings don’t render properly in word, but I’m basically forcing you to read documentation so that you get used to it. It’s actually very clear and readable.

Note that ListIterator’s add method has a void return type. The Collections interface has an add function with a Boolean return type. Don’t confound the two.

### 2. Queue<E>

Queues are characterised by two functions add and remove, that implement FIFO (First In, First Out). Add throws an error if there is no more room, and remove throws an error if there is nothing to remove.

A “gentler” version of add is offer. If it’s not possible to add, null is returned.

Remove has a gentler equivalent called poll.

Inspecting the head element, but without removing it can be done with the command element, which could throw an error.

Gentler equivalent is peek.

public interface Queue<E> extends Collection<E> {  
 */\*\*  
 \* Inserts the specified element into this queue if it is possible to do so  
 \* immediately without violating capacity restrictions, returning  
 \* {****@code*** *true} upon success and throwing an {****@code*** *IllegalStateException}  
 \* if no space is currently available.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Collection#add})  
 \** ***@throws*** *IllegalStateException if the element cannot be added at this  
 \* time due to capacity restrictions  
 \** ***@throws*** *ClassCastException if the class of the specified element  
 \* prevents it from being added to this queue  
 \** ***@throws*** *NullPointerException if the specified element is null and  
 \* this queue does not permit null elements  
 \** ***@throws*** *IllegalArgumentException if some property of this element  
 \* prevents it from being added to this queue  
 \*/* boolean add(E e);  
  
 */\*\*  
 \* Inserts the specified element into this queue if it is possible to do  
 \* so immediately without violating capacity restrictions.  
 \* When using a capacity-restricted queue, this method is generally  
 \* preferable to {****@link*** *#add}, which can fail to insert an element only  
 \* by throwing an exception.  
 \*  
 \** ***@param*** *e the element to add  
 \** ***@return*** *{****@code*** *true} if the element was added to this queue, else  
 \* {****@code*** *false}  
 \** ***@throws*** *ClassCastException if the class of the specified element  
 \* prevents it from being added to this queue  
 \** ***@throws*** *NullPointerException if the specified element is null and  
 \* this queue does not permit null elements  
 \** ***@throws*** *IllegalArgumentException if some property of this element  
 \* prevents it from being added to this queue  
 \*/* boolean offer(E e);  
  
 */\*\*  
 \* Retrieves and removes the head of this queue. This method differs  
 \* from {****@link*** *#poll() poll()} only in that it throws an exception if  
 \* this queue is empty.  
 \*  
 \** ***@return*** *the head of this queue  
 \** ***@throws*** *NoSuchElementException if this queue is empty  
 \*/* E remove();  
  
 */\*\*  
 \* Retrieves and removes the head of this queue,  
 \* or returns {****@code*** *null} if this queue is empty.  
 \*  
 \** ***@return*** *the head of this queue, or {****@code*** *null} if this queue is empty  
 \*/* E poll();  
  
 */\*\*  
 \* Retrieves, but does not remove, the head of this queue. This method  
 \* differs from {****@link*** *#peek peek} only in that it throws an exception  
 \* if this queue is empty.  
 \*  
 \** ***@return*** *the head of this queue  
 \** ***@throws*** *NoSuchElementException if this queue is empty  
 \*/* E element();  
  
 */\*\*  
 \* Retrieves, but does not remove, the head of this queue,  
 \* or returns {****@code*** *null} if this queue is empty.  
 \*  
 \** ***@return*** *the head of this queue, or {****@code*** *null} if this queue is empty  
 \*/* E peek();

}

A Deque is a double ended queue, so it has

boolean addFirst(E element);  
boolean addLast(E element);  
boolean offerFirst(E element);  
boolean offerLast(E element);

E removeFirst();

E removeLast();

E pollFirst();  
E pollLast();

E getFirst();  
E getLast();  
E peekFirst();  
E peekLast();

Apart from all the methods of an ordinary Queue. (In case you forgot, any concrete implementation of Deque, Queue etc. will have the collection interface functions like addAll, iterator, and all those other seemingly obscure methods.)

### 3. Set<E>

Sets are characterised by two things, their uniqueness and their quick membership checks.

Calling add() on the set will return false if the element already exists, and set equality is determined after the sequence of elements is discarded.

Typically sets are implemented with a hash table, so the presence of an element can be checked in O(1). However, this means that the Set is not Ordered. The iterator will just go through all of the elements in the has table, but since it is in the hashed order, the elements themselves are unordered.

It is possible to keep an ordered set, a sorted one in fact, but in order to keep the access times low, this is typically done with a balanced binary search tree. Access times are O(log n).

### Maps

Maps are key value structures which operate on a different set of rules from normal collections. There are 4 main methods of interest:

V put(K key, V value)

V get(Object key)

boolean containsKey(Object Key)

boolean containsValue(Object value)

put adds an entry to the dictionary, and returns null if it was successful. If the Key already exists, then the existing value V is returned instead.

get, containsKey and containsValue take Object type arguments for legacy reasons, and in order to support a broader definition of equality that can equate objects of type K with some other random class. This is a controversial design choice of Java which led to most implementations having to do many type checks in the implementations.

Commonly, we use a hashmap to count the frequencies of events, and this always results in some awkward initialisation conditions:

if k in dict:  
 dict[k] += 1  
else:  
 dict[k] == 1

To solve this, Java provides the getOrDefault(Object key, V defaultValue) method. If the get method says that it doesn’t exist, then the default value is returned.

freq.put(key, freq.getOrDefault(key,0)+1);

Alternatively,

freq.putIfAbsent(key,0);  
freq.put(key, freq.get(key)+1);

Alternatively, Java also gives a very cool merge function:

freq.merge(key, 1, Integer::*sum*);

If the key does not exist, set given value (argument 2) as the value. Else, use the given function (argument 3) to merge the existing value and the given value.

An explicit check is also possible, and here are the relevant functions:

Set<K> keySet();

Collection<V> values();

Set<Map.Entry<K,V>> entrySet();

It is possible to iterate through the keySet and thereby the Map. A more convenient loop would be through the Map.Entry Set.

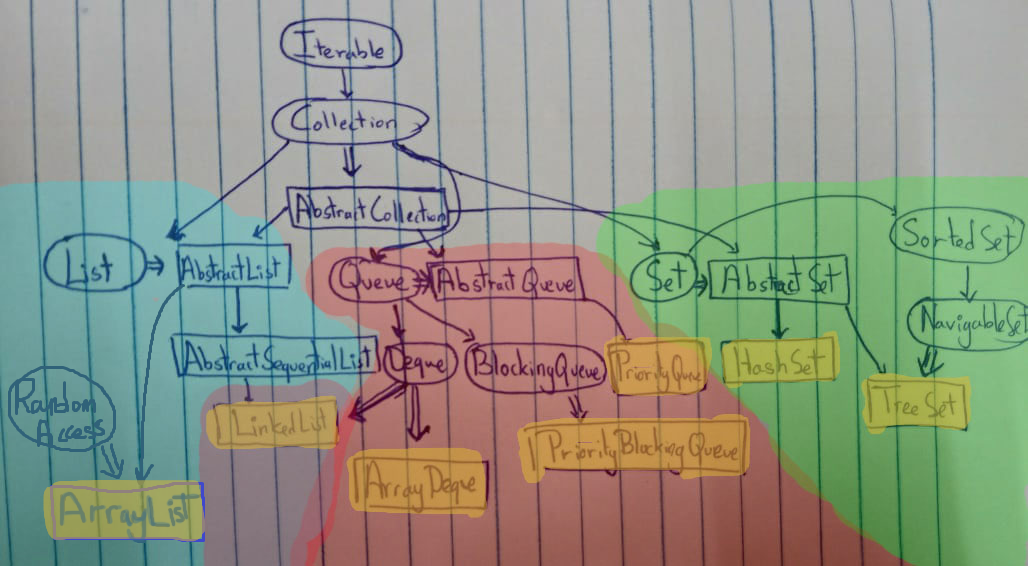
The Map.entry data type has 2 important functions: getKey and getValue.

Concrete implementation: HashMap, TreeMap

keySet() of TreeMap is a TreeSet, for HashMap it’s HashSet.

LinkedHashMap has two way pointers in the storage location, allowing ordered traversal in the order of insertion.

### Summary



(I) Collections

- 4 core methods: add, remove, contains, iterator. Tons of other crazy methods.

- 3 Great children : Set, List, Queue.

1. Lists

- can be RandomAccess or not (sequential access)

- has a separate listIterator method, which can traverse in both directions.

- Concrete random: ArrayList; Concrete Sequential: LinkedList, which is also a deque.

- Ironically, the standard array used does not fall under collections; it is a standalone class called Arrays.

2. Queue

- add , remove only work on tail and head, respectively. Gentler version that does not raise exception: offer and poll.

- Inspection of head is done with element, gentler version is peek.

- Deque has all these 6 methods for both ends.

- Priority Queue’s remove method always removes the highest priority item.

- Blocking Queue’s take and put methods wait until an object is available to take, or until room is available for adding, and then perform the operation. It operates on many threads at once, and is said to be “[Thread safe](https://learn.microsoft.com/en-us/archive/blogs/ericlippert/what-is-this-thing-you-call-thread-safe)”, in some sense.

3. Sets

- add works only if the element doesn’t already exist in the set. Set equality works without considering order. Element presence can be checked very rapidly.

- HashSet uses a hash table to implement set, the most common type.

- A sorted set maintains some ordering; it’s possible to construct a complex hashed set which has double pointers to other elements allowing ordered traversal. The concrete implementation of that is called a LinkedHashMap. One more concrete implementation is the TreeSet which uses a balanced binary search tree to implement SortedSet.

(II) Maps

There are 10 main methods of interest:

- V put(K key, V value) : adds an entry to the dictionary, and returns null if it was successful. If the Key already exists, then the existing value V is returned instead.

- V get(Object key)

- boolean containsKey(Object Key)

- boolean containsValue(Object value)

These take Object type arguments for legacy reasons, and in order to support a broader definition of equality that can equate objects of type K with some other random class. This is a controversial design choice of Java which led to most implementations having to do many type checks in the implementations.

- V getOrDefault(Object key, V defaultValue) method. If the get key doesn’t exist in the map, then the default value is returned.

- V putIfAbsent(K key, V value). Puts only if key is absent in the map.

- V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)

If the key does not exist in the map, set value (argument 2) as the value. Else, use the given function (argument 3) to merge the existing value and the given value. In the generic type of the BiFunction, Arg1 is existing value, Arg2 is new value, and Arg3 is the return type.

- Set<K> keySet();

- Collection<V> values();

- Set<Map.Entry<K,V>> entrySet();

These are commonly used to iterate through the Map. Since all three of these objects have an iterator method in any implementation, a for each loop can be used on them.

The Map.Entry Class has 2 methods of interest: getKey and getValue.

There are 3 important implementations we must be aware of:

HashMap, TreeMap: the keySet of each of these return a HashSet, TreeSet respectively.

LinkedHashMap has two way pointers in the storage location, allowing ordered traversal in the order of insertion.

# Part 7: Features For Flair

## Cloning

It’s possible to create copies of objects with a copy constructor, but what if I told you, there is a much dumber way to do it?

Yes, you heard me right, DUMBER.

Argh, GOD, sometimes, Java just makes me want to slam my face on the keyboard till I break my laptop or my skull, whichever occurs earlier, only so that I never have to see Java again.

Abstract Classes with the names very similar to interface counterparts because default functions didn’t exist way back?

Dumb naming like the Logger interface in one standard package, and a Logger class in another?

weird argument types in map for legacy support?

Not having multiple inheritance, and then introducing interfaces, and then introducing default functions to make a mess anyway?

And now, the primordial beast of all legacy java code, CLONE.

Clone creates a bitwise copy of an object. Thus, even if there are private variables, it doesn’t matter to clone. It goes and copies every single one and zero and makes a new object.

What could possibly go wrong?

Well, there a lot of attributes that are actually stored by reference, meaning that there will be a pointer to a memory location in the object, and that pointer will also get bitwise copied. Now, both these objects point to the same thing, and edits on either will mess with the other.

Disaster.

This is called a shallow copy. To make a deep copy, you must clone every reference variable:

class Employee implements Cloneable {  
  
 public Employee clone() {  
 Employee copee = (Employee) super.clone();  
 Type rvarcopee = copee.referencedvar.clone();  
 copee.referencedvar = rvarcopee;  
 }  
}

Firstly note that the clone variable’s return type is specific to the class, but even so, it is overriding the clone method. It is calling super.clone. Clone is a unique function in this regard: It is allowed to override its parent function even through their signature differs in the return type.

By default, clone is a protected function, so it cannot be used outside its own class, or its subclasses. Its visibility can, however be extended to public if we wish to use in other classes too.

Secondly, because super.clone() (the legendary function that actually makes a bitwise copy) will return an object of class Object, we must typecast it.

To show that a class supports the clone method, it is necessary for it to implement the marking interface Cloneable. (marking = empty, just like RandomAccess)

If clone is not supported by an object, CloneNotSupportedException is thrown, and because every call of clone is capable of throwing this, every call of clone is supposed to be in a try catch block.

When a class implements clone, its subclasses unfortunately inherit this function, so if we call clone on a subclass, we might inadvertently make a copy only deep upto the parent class and shallow on the rest. Because of this, programmers are expected to override the clone function in every subclass and raise a CloneNotSupportedException in the method, if they do not wish to continue the madness.

Java itself actively discourages use of clone in this manner: if you are really going to go for this dumb choice, then it makes you climb a few mountains during which you can reconsider your life decisions.

## Type Inference

For a long period of time, Java did not have type inference. Now, it permits local variables within methods alone to be type inferred. Object fields cannot be type inferred.

Type inference is a tricky thing, especially when you want your compiler to be catching errors.

var x ="Hi" + "Globe";

Here, the data type of the RHS is string, and is propagated to the left hand side. This is the simplest case.

In python, apart from x being defined to be any object, it is valid to call any method on x. If it doesn’t work, the code will just crash on runtime. That’s python’s thing, it just won’t worry about anything until it crashes face first into it, kind of like how I study for my tests.

To catch such a thing before hand, static typing only allows methods strictly possessed by that class. However, if an employee variable is used in a function, and you want to call a manager specific method in the method, it could just be that every time the function is called, you send only manager objects. That is, the more ambitious and welcoming way to typecheck, is to assume that the code is type correct when an assumption needs to be made, and see if the rest of the code conforms to it. This, of course means that the function call cannot be validated purely on the static type any more.

The cost here is doing more and more work on compile time to pre-emptively catch errors. A strict set of static typing rules like in Java helps the programmer catch errors, especially when code reaches unwieldy proportions, without having to do a terrible lot of work to validate the code.

Permitting a limited amount of type inference is just Java venturing a bit farther in that balance between complexity of the compiler and the flexibility in coding that can be gained from it.

Apart from type inferring local variables, Java type infers almost everything about Lambda Functions, and a *lot* of things about generic functions in general with just the calling context. This is incredible to watch in action, as we will find out soon.

## Lambda Functions

- Often, methods need to take functions as arguments. However, this is not allowed, as only classes exist in Java. Thus, the function is housed in a “functional interface”.

Eg: Comparator<T> has one method, compare() which can be used to compare two objects. If a comparator is needed for a function, this will be the argument type.

- Essentially, we have a box, a function, and instead of passing around the box, we put it in a bigger box, called a class and then pass that around instead. Although on the level of principle there is no way to avoid it in Java, it can be averted in code.

- (arg1, arg2…) -> {yada yada return x yaay!} this represents a lambda function. It is a function that seemingly exists in isolation and has no predeclared return type. Such a function can be fed into a method that expects an object implementing a functional interface; the single function associated with that interface is the lambda function. Note that this means that the signatures must match.

- Explicitly, this is equivalent to declaring a class implementing the interface, overriding the abstract method with the function body, and then passing a dummy instantiation of said class into the function.

We will see more of lambda functions in the following sections.

# Part 8: Streams & Optional

Streams are means to handle sequences of arbitrary length, (even infinite) lazily. With the level of Java we have already learnt, you may assume you can understand almost all of the core Java code, but the more you look at Stream source code, the more you realise how much there is left to learn. I could try and go into the details here, but it’s a messy web.

1. Making Streams

- Stream<T> is the type of a stream of objects of type T.

- Stream.of(obj) is a means of converting any objection into a stream; Typically, this is an array. Alternatively, methods meant for streamifying within other classes can be used, such as the .stream method of List<>.

- Stream.generate takes a function argument, and uses the value of that function for each entry of the sequence. For instance, Stream.generate(math::random) is an infinite stream of random numbers.

The argument is of type Supplier.

@FunctionalInterface  
public interface Supplier<T> {  
T get();  
}

- Stream.iterate() is used to create a dependent stream; It also takes a function argument, but this function takes one argument, which is the previous entry in the stream.

Eg: Stream.iterate(0, n ->n+2 ) is a sequence of even numbers. (0 represents the start value)

The argument to iterate is a Unary Operator:

@FunctionalInterface  
public interface UnaryOperator<T> extends Function<T, T> {  
static <T> UnaryOperator<T> identity() {  
 return t -> t;  
 }  
}

Where Function is

@FunctionalInterface  
public interface Function<T, R> {  
  
R apply(T t);  
default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {  
 Objects.*requireNonNull*(before);  
 return (V v) -> apply(before.apply(v));  
 }  
  
 default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {  
 Objects.*requireNonNull*(after);  
 return (T t) -> after.apply(apply(t));  
 }  
  
static <T> Function<T, T> identity() {  
 return t -> t;  
 }  
}

The key feature of a functional interface is that it has *exactly one* abstract function, and only such an interface can accept a lambda function. That abstract function is what the lambda function defines.

Here, the abstract function is apply; The rest are irrelevant. When a function n -> n+1 is fed into unary operator, it basically defines T apply (T n){return n+1}, but then the UnaryOperator class will be used *raw*, so somebody needs to find T. To do that, the type of the seed is used to infer T. In (0, n->n+2), since the seed 0 is an int, T is Integer.

When we gave math::random to Supplier, T would’ve been fixed as double by the return type of the function.

- Stream.iterate can also be used with 3 arguments, the second being a condition that determines when the stream must be terminated. This is a boolean function that takes n as the argument.

Eg: Stream.iterate(1, n -> n<100, n -> n+2) will give odd numbers upto 99. When a new number is generated, it must pass the check of n<100. If it fails, the stream is halted.

@FunctionalInterface  
public interface Predicate<T> {

boolean test(T t);  
default Predicate<T> and(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) && other.test(t);  
 }  
  
default Predicate<T> negate() {  
 return (t) -> !test(t);  
 }  
default Predicate<T> or(Predicate<? super T> other) {  
 Objects.*requireNonNull*(other);  
 return (t) -> test(t) || other.test(t);  
 }  
static <T> Predicate<T> isEqual(Object targetRef) {  
 return (null == targetRef)  
 ? Objects::*isNull* : object -> targetRef.equals(object);  
 }

@SuppressWarnings("unchecked")  
 static <T> Predicate<T> not(Predicate<? super T> target) {  
 Objects.*requireNonNull*(target);  
 return (Predicate<T>)target.negate();  
 }  
}

The @SuppressWarnings annotation is used to prevent a warning from being thrown.

Here, an unchecked cast is being made. This is a static function, so the class is used to invoke the method, not some object. (Predicate.not(mypredicate).

Two things seem fishy: 1. What even is T? isn’t it being used as a raw class?

2. if the return is to be casted as Predicate<T>, and the return type of target.negate is Predicate<Q>, where Q is the class of target, why is the static type of the argument target, <Q super T>?.

The answer to both of these questions is that there is some type inference going on here.

1. Predicate<String> = Predicate.isequal(myObject);

The T is determined by the *calling* context. The Predicate can consume any string and check if it’s equal to myObject.

2. Predicate<String> = Predicate.not(myPredicate);

Here, T is String, and myPredicate typically consumes something that’s a superset of String, so passing Strings to it shouldn’t be an issue. Now, taking the not of this, we must cast it into the Predicate<String> type. We know that in this case, a Predicate<Q> can always be cast into Predicate<T extends Q>, but the cast is “unchecked”, which raises a warning. Since we know that there is no concern in this case, we simply suppress the error.

Note that I am deliberately not discussing actual Stream code, but only the much simpler definitions of classes used in the Stream methods.

2. Modifiying streams

- the .filter method takes a Predicate function as the only argument. Only elements that result in true are retained in the stream. Filter will go through every element in the stream.

Eg: Stream<String> longwords = wordlist.stream().filter(w -> w.length() > 10)

The calling context of strings being inside the wordlist.stream is what sets the value of T in the Predicate used.

- the map function maps each entry to a new value.

Eg: Stream<Integer> squares = Stream.iterate(1, n-> n+1).filter(n -> n<100).map(n->n\*n)

Throws an error when it runs out of memory! Filter will look at every element; if you wanted to terminate the stream, use the 3 arg iterate, or equivalently, takeWhile() (don’t do this, just to be more efficient; that’s the point of the 3 arg iterate). Using a limit method will allow the code to actually finish executing, even when filter is used on an infinite stream.

- takeWhile() takes a Predicate function; stream keeps evaluating elements till the function returns false. That false entry is discarded. dropWhile does the opposite.

- limit(n) takes only the first n entries of the stream.

- max and min take a Comparator function and return a single value from the stream.

- flatmap is an alternative to map that collapses list brackets, and returns a flat stream instead of a stream of lists or something of the sort. (this makes sense only if the function given to map returns a collection)

- any stream that has been filtered can potentially be empty. An empty stream is still a valid stream, but if a function like max is called on it, the result may be null. To account for this, an “Optional” type is used for the return value.

- Languages tend to handle scenarios where an optional type is needed with a null pointer. Null can be considered implicitly to be of any type, so any object might end up being null. This is true in java as well, and calling regular methods on null will result in a NullPointerException on *runtime*.

- Java also offers a much cleaner fashion of dealing with nulls, which is the Optional<T> Type. It’s a box of type T, which can either have something in it, or just be an empty box. An empty box is very different from null; we can do a lot of cool things with it. I’ll discuss it in a little while.

3. Collecting values from Streams

- mystream.forEach(x -> func) iterates through the stream.

- To get an array out of a stream, use mystream.toArray(). This will return an array of **Object**s by default, and to change the type of the element in the array, pass an array constructor into the toArray function, for example, String[]::new.

- mystream.collect(factory function) makes a collection out of a stream. Any collection of choice can be made with an appropriate factory method passed into it. Such a method will implicitly create an object and populate it with the entries in mystream. Eg: Collectors.toList(); More generally, mystream.collect(Collectors.toCollection(MyCollection::new)) can be used.

Please Ignore the following two pages. They’re my attempt at understanding the collect function. However, it slowly became incredibly obvious it wasn’t worth the effort, since its heavily tied to the stream definition methods, which I already gave up on after a lot of effort. The problem is that there a very large number of interconnected methods and classes, a lot of which are abstract, making it unclear how exactly things work. Maybe actually instantiating an object, and stepping into the rabbit hole with the debugger can help, but I’ve gone sufficiently insane already.

What I did learn from my venture into the docs, is 2 new pieces of syntax:

1. (String… strs) The last argument to a function can end with 3 dots, denoting that strs is always an array, but the function can be invoked like (‘a’, ‘b’, ‘c’). It’s a variable length argument.

2. Record classes (Records are a feature introduced in Java 14, and refined in 15).

“While working on Java projects, we, as developers, often write *service* classes, *security* classes, or any other basic classes. These classes are functional by nature. Similarly, programmers often write classes for the sole purpose of carrying data. For instance, suppose, a client requests some data from the server such as an **id** and **name** of a person and the server responds back with the appropriate data. Since *everything* is an object in Java, there must be some class that carries the data. The server responds back with the object of the class to the client. Note that the sole purpose of the object is to carry the data from the server to the client.

Now, writing such a data class, even if it may be a simple POJO [*Plain Old Java Object*], includes a lot of boilerplate code, such as private fields, constructors, getter and setter methods, **hashCode()**, **equals()**, and **toString()** methods. A simple carrier class becomes heavy with a lot of unnecessary code due to the verbose nature of the Java language. These downsides led to the introduction of a special type of class called **record**. This class aggregates – or holds – a group of values without having to write boilerplate code and acts as an efficient carrier of data objects.

In fact, developers can manage everything without the record classes as we have been doing so long. The record class redefines data carrier classes to another level both in terms of convenience and efficiency.”

If we want a DTO (Data Transfer Object) Employee, rather than defining a full class for it, we can simply do

public record Employee(int id, String firstName, String lastName) {}

instead of a full class definition.

Behind the scenes, a record object has default compiler declared methods for equals, hashcode, toString etc.

-x-

Useless segment:

This is the toCollection method:

public static <T, C extends Collection<T>>  
Collector<T, ?, C> toCollection(Supplier<C> collectionFactory) {  
 return new CollectorImpl<>(collectionFactory, Collection<T>::add,  
 (r1, r2) -> { r1.addAll(r2); return r1; },  
 *CH\_ID*);  
}

where CollectorImpl is just a record implementation of the Collector Interface.

record CollectorImpl<T, A, R>(Supplier<A> supplier,  
 BiConsumer<A, T> accumulator,  
 BinaryOperator<A> combiner,  
 Function<A, R> finisher,  
 Set<Characteristics> characteristics  
 ) implements Collector<T, A, R> {  
  
 CollectorImpl(Supplier<A> supplier,  
 BiConsumer<A, T> accumulator,  
 BinaryOperator<A> combiner,  
 Set<Characteristics> characteristics) {  
 this(supplier, accumulator, combiner, *castingIdentity*(), characteristics);  
 }  
}

1. “Stream” is an interface, not a class. So, .collect doesn’t have an implementation there. .collect accepts a Collector and returns a Collection.

2. The Collector needed by .collect is typically obtained from the toCollection functions defined in the “Collectors” class. These functions return the record implementation of the “Collector” interface.

3. The Collector<T,A,R> interface relies on 3 interfaces: Supplier, BiConsumer and BinaryOperator. None of these have any implemented functions of relevance, since they are interfaces.

4. The streams typically made with .of, .iterate or .generate belong to a class called ReferencePipeline.

-x-

- It’s possible to get all the descriptive statistics about a stream in one go using summarizingInt. It’s used in a somewhat peculiar way: mystream.collect(Collectors.summarizingInt(x -> func))

The function is meant to convert the given stream into an integer stream, so as to use summarizingInt on the stream. This returns an object of type IntSummaryStatistics, which has the min, max, avg, count and sum, all accessible with getters.

Similarly, there exists the Double and LongSummaryStatistics classes with the corresponding summarizingDouble and Long methods in Collectors.

- groupingby partitioningby and toMap with a merge function

### The Optional Class

Making Optional Types

- When a functions return type is specified to be Optional<Something>, returning Optional.empty miraculously auto types to the correct type.

- Optional.of(value) returns the box with the item.

- Optional.ofNullable(value) transforms a null value into the empty optional.

Operating on Optional

- .orElse(value) returns value in box. if box is empty, the value argument is returned instead.

- .orElseGet(function) returns value if present, else calls function to generate the returned value.

- .orElseThrow(throwable object *Supplier*) Eg: ExceptionNameHere::new; when the function is invoked it returns a pointer to an object of a class that extends Throwable.

- .ifPresent(v -> do something with v) Eg: add v to a list optval.ifPresent(results::add)

- .ifPresentOrElse(v -> do something with v, () -> do something in absence of v)

- .or(Optional *Supplier*) if value is not present, return the optional obtained from the supplier. The optional on which the method is invoked is unchanged, as is the case in most of the above methods.

- .map((v) -> func) the value is mapped, if no value, the optional remains empty. The outcome of map is an optional.

- .flatmap(func). If the return value of the function is in itself an Optional, map will store an Optional containing an Optional, a rather dumb thing to do. Flatmap removes the extra layer, and stores only the value of the optional returned by the function.

# Part 9: Concurrent Programming

This is a very fun topic.

In order to utilise the computational resources available to us most efficiently, or in order to implement something that is fundamentally parallelised, several threads simultaneously work on a task or a set of relevant tasks.

As an analogy, if one thread is downloading a large file, and another handles the browser GUI, when we cancel the download, the first thread must be interrupted; it cannot say, hang on let me finish what I am doing and then listen to what you have to say. This is an example of a scenario where 1 thread is simply not enough.

Another example: a ticket booking system. If there is only one seat left in the movie theatre, two people must not be able to book it simultaneously. Same goes for products on ecommerce sites.

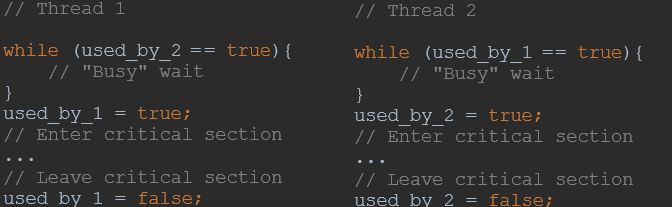
Despite the fact allowing multiple users to simultaneously access and more importantly, make edits on the database, can give rise to a conflict, these applications are fundamentally multi-threaded. “Just one person can interact with the database, everyone else can wait their turn” is a dumb policy.

The solution must involve some sort of blocking: if some edits are being done by one thread on a variable, then that variable cannot be accessed or edited by any other process.

That way, if people are requesting disjoint parts of the database, each of them making edits causes no harm. Only when two threads request a common variable, one of them needs to be blocked, if not blocking can give rise to inconsistent states.

A section of code where shared variables are accessed by a thread is called a Critical section. Rather than making entire threads work only one at a time on a database, they can be allowed to work together, as long as no two critical sections execute simultaneously. (The threads face “mutual exclusion”)

Attempt 1 at mutual exclusion:

Set a flag that says “1 is using critical resources” or “2 is using critical resources”:

The issue with this, is that this does not “Lock” the resources as exclusive for that thread before it **decides** to enter into the critical section. The problem with that is that the following sequence of steps can occur:

0. used\_by\_1 = false, used\_by\_2 = false

1. Thread 1 executes: while(used\_by\_2\_==true){} // bypassed

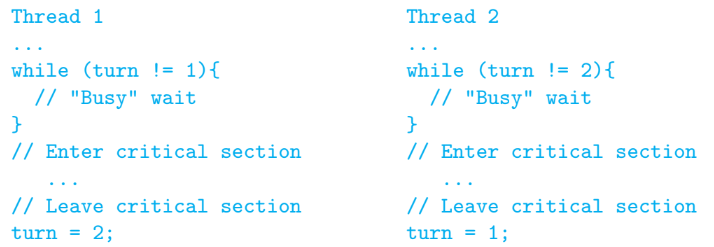
2. Thread 2 executes: while(used\_by\_1==true){} // bypassed

3. Thread 1 executes: used\_by\_1=true;

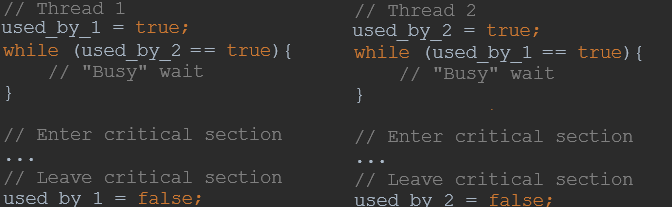
4. Thread 2 executes: used by\_2 =true;

now both of them are accessing critical resources.

Okay, so one must be assured that if the while condition allows a thread to proceed that must **necessarily imply** that thread and that thread alone is able to pass that stage which occurs in every thread. Here, used\_by\_1 and used\_by\_2 are independent variables, so there is no reason for only one of them to be true at any given time.

Attempt 2: Let’s make used\_by into turn, which takes up values 1 or 2. Since turn can only be 1 or 2 at any given time, both processes cannot begin. We have established “Locking”.

The problem here is that there is mutual dependence. One thread cannot independently execute. It *requires* the other thread to take up its turn and give back its turn. If only one thread is running, or worse, if one thread crashes, one thread is left to starve due to the other’s absence. (“Starvation”)

Attempt 3: Going back to attempt 1, if we set the variable used\_by *before* the while loop, and thereby in a rather scammy way, we can establish Locking.

In a sense, you just call dibs on the critical data and everyone else can wait. If someone is already using it, you just wait till they let go. In this case, used\_by\_1 is a bad variable name, it must instead be requested\_by\_1 since it means “1 has requested critical resources, anyone else trying to use them must not execute.”.

It’s immediately obvious what will go wrong here, especially if we think of n threads: multiple threads can call dibs at the same time, and all of them will just end up stuck in the “wait” state. This is a “Deadlock”.

Okay… Are we thinking about this wrong? Let’s make some dumb analogy and see where it goes. Let’s say there is a restroom with multiple doors.

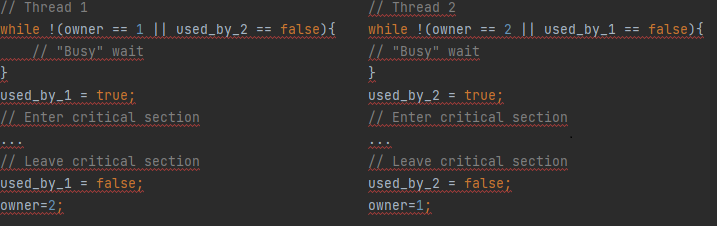
Attempt 1 is saying, the moment someone enters, they hit a button that says “1 is in here”, but the problem is that 2 can enter the room before 1 hits the button, and it’s a shitshow.

Attempt 2 is saying that the restroom explicitly belongs to either 1 or 2, and every time 1 leaves the room they set the ownership to 2, and if one doesn’t own the room they don’t enter it. The problem is obvious here, 1 can die inside the toilet, or just go on vacation to Hawaii.

Attempt 3 is saying people can come and plaster their names on the restroom doors to call dibs, and if someone else’s name is plastered, they don’t enter. When they exit, they take off the plasters. Problem is that multiple people can plaster their names, and nobody enters.

(People dying in the toilet is always a problem, here and always. It’s fixable, in this case though, the exception handler just needs to take off the plasters and take them to the burial service. In the last case, the 2 people just couldn’t live without each other.)

For exactly 2 people, it’s possible to *combine* attempts 2 and 3:

Ownership can act as a tiebreaker in attempt 3, where nobody enters.

1 enters if either if they own the room, or if two hasn’t plastered the walls, indicating nobody is inside.

1 still plasters their name on the doors when they enter and removes it on exit, and changes ownership to 2 when they leave. But, because 1 can enter as long as there are no plasters, they can keep using the room even when 2 owns it. 2 won’t enter as long as they see the plasters; The ownership is merely a tiebreaker.

We took attempt 3 and solved “deadlock” and coincidentally, “starvation” also got solved.

Since ownership changes for more than 2 people isn’t obvious, it’s not obvious how to generalise this to n people. When there are n people, the room being owned by a dead thread, causes deadlock too; the situation is a little bit more complex.

For 2 people, this is called “Peterson’s algorithm”, and although it’s a very clever and interesting algorithm, especially because it doesn’t make any demands of the computer, its generalisation to n threads is not of practical utility.

For example, Attempt 1, simply putting an “occupied” flag on every door is a valid solution, if the computer can guarantee that the moment someone enters, instantaneously the flags go up, and nobody can possibly read the state as unoccupied and enter. Such a demand ***cannot*** be met by some clever set of lines of code, ***no matter what we do.***

Dijkstra came up with the idea of using “semaphores”, railway signal flags, that are put up or down to show that passage is allowed or blocked in concurrent programming, but semaphores are something we must demand of the system as a **fundamental** function we can invoke in code.

In summary, using a semaphore, or a Lock – Unlock system for the door of the room is a very simple solution, and is the one in practice. However, it requires that the fairy godmother grant us semaphore functions; One of the cleverest algorithms that can establish a “thread-safe” system, without any magic is Peterson’s algorithm.

The most intuitive n process mutual exclusion algorithm is Lamport’s Bakery Algorithm, also known as the food court queue. The counter gives each thread a token, and then the tokens are called one by one to get access to the critical resources.

However, not only are these algorithms complex, they are a pain to repeatedly to use in code. Semaphores are a much more convenient and powerful tool, and in order to implement them, we must define an “atomic test and set” operation at the CPU/transistor level.

Define class Semaphore with 2 methods: Passeren() and Vrygeven() (meaning: dijkstra wasn’t English)

// Common   
Semaphore S = new Semaphore();  
  
// Thread 1  
  
S.P()  
// Enter critical section  
...  
// Leave critical section  
S.V()

P() **atomically** executes the following

if (S > 0) {decrement S;} // one step

else{wait for S to become positive;}

V()executes the following

increment S;

if (there are threads waiting for S to become positive)

wake one of them up; //choice is nondeterministic

Observe S can only go between +no of permitted simultaneous operations and 0. The “occupied” flag is a 1-0 thing. If it’s unoccupied, decrement and enter. When exiting, increment. Sleeping threads are woken up if any are around.

This S is a generalisation of the occupied flag to n rooms. If S=n, n rooms are free. A process can decrement and enter. If no rooms are free, processes must wait.

Java has a public Semaphore class which we can use manually in this fashion to establish mutual exclusion, but doing this every time can be a bit exhausting. A lot of the job is thus handed off to Java. Java defines a “monitor” for every class. If a bunch of methods are tagged synchronized, then only one of them can execute at any given time.

You may wonder, why a monitor for every class? The only ones running in separate threads are ones that implement Runnable right? Not exactly; Those can call upon other objects. Basically, because multithreading is a thing, any 2 processes can potentially run at the same time. Two or more methods within the same class can be protected from conflict by simply adding the synchronized keyword in the method definition.

public synchronized void ...

In summary:

1. Java allows for multithreading, which, if used frivolously can lead to unexpected behaviour or inconsistent states. The cause of the problem is the fact that some critical components are meant to be accessed by exactly one process at once.

2. Java resolves this with the introduction of monitors; multiple methods can be tagged with the keyword synchronous, and java will see to it that no two of them ever execute simultaneously; They are made to wait their turn and execute one after another.

3. From a theoretical point of view, there are algorithmic solutions to the Mutual Exclusion (Mutex) problem: Peterson’s Algorithm for 2 processes, and Lamport’s Bakery Algorithm for n processes. These are very interesting to learn of and make no demands on the system, and as a consequence are not the most powerful things practically usable. It is possible to guarantee an atomic test and set operation by working at a very low level, and once this is done, Semaphores can be used for mutex.

4. Java allows the use of a Semaphore class, which is much more flexible: For instance, if we may wish for critical sections to be segments of code rather than entire methods; monitors ask that we just separate out the critical segments into their own methods, which may not always be convenient.

5. Another Java class that can establish mutex is ReentrantLock, and it is very similar to Semaphore, but the two have slightly different use cases.

6. wait(), notify() and notifyall() are used to implement an internal queue in Java.

7. BlockingQueue and ConcurrentMap are powerful thread safe data structures.

-x-

The Semaphore class just increments a counter when resources are released, and decrements it when they are taken up. The thing is, it doesn’t check for a max resource limit. It’s possible to do S.release() without ever calling S.acquire(), so the max no of parallel accessors can be *increased.* If this is undesirable, as it typically is, then it’s just upto programmer discipline. C++ lets you set a limit by the way.

ReentrantLock uses methods lock and unlock, and it works on the same principles. It decrements on unlock and increments on lock, and it has a max count. There is apparently some nuanced difference between ReentrantLock and Semaphore, but I cannot quite understand it.

That aside, there is one more fascinating problem regarding sequencing, that adds a layer on top of mutex: If a series of transactions occurs:

Transfer 500 from i to j; Transfer 400 from j to k.

This will always succeed as long balance(i) >= 500. However, this can fail if the sequence is altered, and balance(j) < 400.

When an action is performed and it cannot be completed, you may not necessarily want to declare it to be a failure right away. You might want to suspend this action and see if some other process comes in and saves the day. Maybe you can wait for saviour for a fixed amount of time, say, 1 min and then declare the task a failure.

In this model there are 2 queues. One: There is an “external” queue waiting to access the critical resources for the first time, blocked by monitors or an equivalent. Once the process enters the critical room, it can choose to wait longer. But if it just waits there, it will not release the resources, so it must wait in a separate “internal” queue, which is woken up every time something noteworthy happens.

Waking up all the internal queue members for every transaction is a valid, but inefficient solution. Waking it up only if the concerned accounts are involved is a much better strategy. To establish this, there can be multiple internal queues, each waiting for a different thing to happen.

Further, there are 3 possible strategies adopted when internal queues are in use: A thread, say a successful transaction between x and y, (x gives y money) can go and wake up all the processes who seek to take money from y. Then the successful transaction can die peacefully. This is choice 1: notify and exit, where the notifying process instantly dies on notifying.

Choice 2: notify and wait – The notifying process swaps roles with the waiting process. This is a very rarely used choice. Choice 3: notify and continue – This is used in java. The process wakes up the concerned threads and says “Just a sec, wait for me to be done, something good is going to happen, stay tuned.” This means that if there code after the notify line, they will execute before the next process comes in.

Java allows the use of the wait(), notify() and notifyall() methods within synchronous methods to implement an internal queue. A thread that has been woken by notify will take control before a thread that had requested the resources before the notify call.

# Part 10: Farewell

Well.

Here we are.

Java is a very interesting and powerful OOP language. It’s far more rigid than I would like, but it is this rigidity that prevents a large number of problems in the later stages of a project. Java forces the programmer to think about certain problems right away when coding, and then forces the programmer as much as it can to prevent a bad solution from looking appealing.

Here is what I find to be the basis on which Java is built:

Privacy, which is used to make APIs that are much easier to maintain and extend in the long run. Polymorphic coding, which allows general, resilient and powerful code; (Interfacing facilitates this structural polymorphism, and Generics enable the programmer to write complex yet generalised code while maintaining code integrity at the compilation stage.)

These fundamentals are sufficient to understand almost all of Java. Beyond this, we analysed the Inbuilt Collections and Map structures, which form the basis of most practical coding. Logging, and exception handling are built with the same object oriented mindset, and are elegant in their own way.

Finally, we looked at some important intermediate features: Lambda functions, Streams and Concurrent programming, and some minor ones: Cloning, Optional classes and the Reflection package. I didn’t bother discussing Input and Output streams, I trust that it will not be too hard to just look up the relevant syntax should we need to use files.

Most of what has been discussed in this doc are features of Java 8. Java is presently in version 20. Naturally, we have only seen the tip of the iceberg, and there are far too many features to look at, should we try to be comprehensive. Such an endeavour is futile. I think we are now equipped to learn what is necessary in order to work with a codebase, and understand the spirit of Java.

I don’t expect to be using Java for any of my personal coding; That would be in python; I don’t expect to be using it for competitive programing either; That would be in C++. But I do expect that should I encounter a large project underway or otherwise, Java is a tool at my disposal, and that what I’ve learnt of Java will help me write better OOP code in general.

And perhaps that makes all the difference.