PBCST304: Object Oriented Programming

Contents

Chapter 1

Introduction to Java

Java is a general purpose, object oriented, platform independent programming language that forces you into the OOP paradigm. Every java program consists of a class (or multiple) classes that contain various components such as functions (called methods in OOP), constructors, interfaces, variables etc.

1.1 Structure of a Java Program

A minimal structure of a java program includes

1. Package declaration (optional)
2. Import statements
3. Class definition
4. main method
5. Statements inside main

For example

```
class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Where,

1. 'class Main' defines the class named Main. Typically this means the file name is Main.java
2. 'public static void main(String[] args)' is the entry point for the program
3. System.out.println() prints to the standard output.

1.2 Java Runtime Environment (JRE)

The JRE contains all the tools required to execute a java program. It consists of:

1. The Java virtual machine (JVM)
2. Core libraries
3. Supporting libraries

Note: The JRE only runs files, it does not compile them.

1.3 Java Development Kit (JDK)

The JDK contains all the tools required to compile and run files.
It includes

1. The JRE (as mentioned before)
2. The JVM (which is a part of the JRE)
3. Java compiler (javac)
4. Other tools like jar, debugger etc.

JDK is essential to compile and run java programs.

1.3.1 Run a java program from the command line

First save the file with a .java extension. Then

1. Compile using:

   javac file.java
2. And then run using

   java file

Notice how there is no file extension when running the program?

1.4 The Java Compiler

Usually invoked as javac, it is a program that converts java source code into intermediate byte-code (.class files)

The bytecode produced by javac is platform independent which means it can run anywhere so long as a JVM is present.

This approach follows java's motive of 'Write once, run anywhere' (although we all know that's not true lmfao). The code isn't converted directly to some CPU-dependent language like assembly and rather, the bytecode can be interpreted by the JVM to run on any platform.

1.5 Java Virtual Machine

The JVM is the engine that runs java bytecode. It does not execute java code by itself, it only executes the bytecode generated by javac.

The JVM contains the following components:

1. Class loader: loads classes into memory dynamically.
2. Bytecode verifier: to prevent any illegal or unsafe bytecode.
3. JIT compiler (interpreter): what runs your code
4. Runtime memory areas: method area, heap, stack, PC registers, native method stack.
5. Garbage collector: (unlike C) to do automatic memory cleanup

1.6 Primitive Datatypes

A primitive datatype is the most basic form of data values. They are not objects and they store their values directly in memory. They are also fast, lightweight and are used for simple low level operations

In java, there are 8 primitive datatypes

Type	Size	Description	Default Value
byte	1 byte	(-128 to 127)	0
short	2 bytes	(-32,768 to 32,767)	0
int	4 bytes	-2,147,483,648 to 2,147,483,647	0
long	8 bytes	very large integers	0L
float	4 bytes	decimal with single precision	0.0f
double	8 bytes	decimal with double precision	0.0d
char	2 bytes	a single unicode character	'\u0000'
boolean	1 byte*	true/false	false

* depends on the implementation of the JVM

1.6.1 Characteristics:

1. Values are typically stored in stack memory
2. They hold actual values and not references
3. Not objects
4. Faster access and easier to manipulate.
5. Have no methods (eg: int.length() is not possible)

1.7 Wrapper Classes

Every primitive datatype has a corresponding wrapper class in the java.lang package. These are objects that encapsulate (explained in Ch 2) primitive values.

They exist primarily because sometimes you need an object rather than a value. For example, collections like ArrayList cannot store primitive values; Wrapper classes have methods that are often useful like conversion, parsing etc.; and finally nullability which means, an Integer can be null but an int cannot.

Below are the wrapper classes of their corresponding primitive datatype

Primitive	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

This leads us into our next concept

1.7.1 Autoboxing and Unboxing

Autoboxing refers to converting a primitive datatype into its corresponding wrapper class.

```
int x = 10;
Integer y = x; // autobox
```

Unboxing is the opposite. Converting an object into a primitive datatype

```
Integer y = 10;
int x = y; // unbox
```

1.7.2 Some differences between primitives and wrappers

Feature	Primitive	Wrapper
Memory	Mostly stack	Heap
Speed	Faster	Slower
Nullable	No	Yes
Methods	None	Many utility methods
Use in collections	No	Yes

1.8 Casting

Type casting refers to converting one datatyope into another. In java, this is mainly done with primitive types and objects.

There are 2 types of casting

1. **Explicit casting**: also known as **narrowing**, it is the conversion from a larger datatype to a smaller one.

   The name comes from the fact that a programmer will have to manually do the typecast and risk any potential data losses.

   ```
   double d = 9.78d;
   int x = (int) d;
   ```

   the value of x is 9 and the decimal part is lost.

2. **Implicit casting**: also called **widening** is the conversion from a smaller datatype to a larger one.

   The compiler often does this automatically and there is no data loss involved. The JVM handles it

   ```
   int x = 10;
   double y = x;
   ```

   Notice how (double) isn't required here? This is because the compiler will automatically handle it so you need not include that in the code.

I previously mentioned that casting can also be done on objects. Here are a few examples

1. Upcasting (implicit):
   Parent → Child

```
class Animal {}
class Dog extends Animal {}

Dog d = new Dog();
Animal a = d; // upcasting
```

2. Downcasting (explicit):
   Child → Parent

```
Animal a = new Dog();
Dog d = (Dog) a; // downcasting
```

   The (Dog) is required in this case otherwise you will get a ClassCastException.

```
Animal a = new Animal()
Dog d = (Dog) a; // runtime error
```

During arithmetic expressions, java will automatically convert smaller datatypes into larger ones
eg:

```
byte a = 10;
byte b = 20;
byte c = (byte) (a+b);
```

(the cast here is because 10+20 returns an int)

1.9 Arrays

An array is a collection of elements of the same datatype stored in contiguous memory.

Properties

1. Fixed size
2. Zero indexed (i.e. starting index is 0)
3. Can be multidimensional (2D, 3D etc.)
4. Elements can be primitives or objects

An array (one dimensional) is initialized as

```
// declaration + allocation
datatype [] variable = new datatype[size];

// declaration + initialization
datatype [] variable = {/*Some items here*/};
```

The elements of an array are accessed using their index

```
// accessing
System.out.println(array[index]);

// assignment
array[index] = value;
```

1.10 Strings

A string in java is not a primitive datatype but rather an object. It is created from java.lang.String class.

Properties

1. Immutable: value cannot be changed after creation
2. Stored in string pool (for literals)
3. Contains many built in methods

Creating strings

```
String s1 = "hello world"; // literal
String s2 = new String("Hi"); // object in heap
```

Some common string methods (built-in) are:

```
s.length();
s.charAt(index);
s.toUpperCase();
s.substring(start, end);
s.equals(another_string);
```

1.11 Vectors

A vector is part of the java.util package and is very similar to an ArrayList. However it has the following differences

1. It is synchronized (thread safe)
2. It can grow and shrink dynamically
3. Due to synchronicity, it is slightly slower than ArrayList.

```
import java.util.Vector;

Vector<datatype> v = new Vector<>(); // initialization

v.add(element); // insertion

x = v.get(index); // accessing elements
```

And there are other methods such as size(), remove(index), contains(element), etc.

1.12 Operations

Java has several category of operations. They allow you to do calculations, comparisons, assignments and logical operations.

1. **Arithmetic operators**: For basic mathematical expressions. Java has $+, -, *, /, \%$
2. Bitwise operators: to work on bits. usually used in low-level programming. In java there are: $\&, \hat{}, \sim, <<, >>,$ and $>>>$ (unsigned right shift).
3. Relational operators: used to compare values and returns a boolean value. Java has $==, !=, <, >, <=, >=,$
4. Boolean logical operators: used for boolean conditions. In java: $\&\&, ||,$ and !
5. Assignment operators: used to assign values to variables: $=, +=, -=, *=, /=, \%=,$

1.12.1 Ternary Operator

Used as a shorthand for an if-else statement

Syntax:

```
(condition) ? value_if_true : value_if_false;
```

1.12.2 Precedence of Operators

Operator Name	Symbol
Postfix	a++, a−−
Unary / Prefix	++a, −−a, −a, !a, ∼a
Multiplicative	$a * b, a/b, a\%b$
Additive	$a + b, a - b$
Shift	$<<, >>, >>>$
Relational	$<, <=, >, >=$
Equality	$==, !=$
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|
Logical AND	&&
Logical OR	||
Ternary	? :
Assignment	$=, +=, -=, \ldots$

1.13 Control Statements

Control statements determine how the program flows, what gets executed, how many times, etc.

There are 3 major categories

1. Selection statements
2. Iteration statements
3. Jump statements

1.13.1 Selection Statements

Used to choose between different paths based on conditions.

1. if statement: block executes only if condition is true

```
if (x > 0) {
    System.out.println("Positive");
}
```

2. if-else: if the condition is false, it executes code in the else block

```
if (x > 0) {
    System.out.println("Positive");
} else {
    System.out.println("Negative");
}
```

3. if-else if-else: for multiple conditions

```
if (x > 0) {
    System.out.println("Positive");
} else if (x == 0) {
    System.out.println("Zero");
} else {
    System.out.println("Negative");
}
```

4. switch statement: when there are many possible values for a single variable, use this instead

```
switch (day) {
    case 1:
        System.out.println("mon");
        break;
    case 2:
        System.out.println("tue");
        break;
    default:
        System.out.println("invalid");
        break;
}
```

1.13.2 Iteration Statements

Also called looping statements, these are used when we need to execute one part of the code multiple times. There are 3 different kinds of loops in java.

1. for loop: for a finite range of values

```java
for (int i=0; i < n; i++)
    System.out.println(i);

// or, if you have an array
for (int num: arr)
    System.out.println(num);
```

2. while loop: when number of iterations is not known

```java
while (count < 5) {
    count++;
}
```

3. do-while loop: while loop but the condition is checked at the end of each iteration instead of the beginning which means the loop is guaranteed to run at least once even if the condition is initially not met.

```java
do {
    x++;
} while (x < 10);
```

1.13.3 Jump Statements

Jump statements are used to transfer control of the program from one point to another. There are 3 keywords to jump in java

1. break: exit a loop or switch

```java
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
}
```

2. continue: skip the current iteration and move onto the next one

```java
for (int i = 0; i < 5; i++) {
    if (i == 2) continue;
    System.out.println(i);
}
```

3. return: to exit a method. May return a value

```java
return x + y;
```

1.14 Command Line Arguments

when you execute a java program from the terminal, you can also pass some additional values that get stored in the String[] args part of the main method. For example:

```
public class Test {
    public static void main(String[] args) {
        System.out.println("First arg: " + args[0]);
    }
}
```

Run: java Test Hello

Output: First arg: Hello

All arguments are recieved as strings so further conversion is required before you can use them in expressions.

1.15 Variable Length Arguments (VLA)

VLAs in java are defined using ellipses (...). Example:

```
void printNums(int... nums) {
    for (int n : nums) {
        System.out.println(n);
    }
}
```

However there are some rules:

1. Only one VLA is allowed per method
2. And it must appear as the last argument in the arguments list of the method.

1.16 Abstract Classes

Abstract classes are a a way to do data abstraction in java. It is a restricted class that cannot be instantiated (used to create object). To access it, you must instantiate a non-abstract class that inherits from it.

An abstract class can contain:

1. Methods
2. Abstract methods: declarations only, no body allowed in the abstract class. The body for the method must be provided in the subclass.
3. Instance variables
4. Constructors
5. Initialization blocks

It is used as a partial form of abstraction since you can still define normal methods and constructors etc. inside an abstract class.

1.17 Interfaces

Interfaces are another way to do data abstraction. Unlike abstract classes, an interface is totally abstract meaning it can contain only:

1. abstract, default, static or private methods
2. Cannot be instantiated or have constructors, initialization blocks.

In essence, it's a more restricted form of an abstract class that provides for total abstraction when needed

1.17.1 Key Differences Between Abstract Classes and Interfaces

Some of the differences are listed below:

Property	Abstract Classes	Interfaces
Definition	Cannot be instantiated. It contains both abstract (no implementation) and concrete (with implementation) methods	Specifies a set of methods a class must implement. Methods are abstract by default.
Implementation	Can have both implemented and abstract methods	Can only have abstract methods. Java 8 can also have default and static methods.
Inheritance	A class can inherit from only one other abstract class	A class can implement multiple interfaces.
Access Modifiers	Methods and attributes can have any access modifier (public, protected, private)	Methods and attributes are implicitly private.
Variables	Can have member variables (final, non-final, static, non-static)	All variables are implicitly public static final (so constants).

Chapter 2

Some Basic OOP Concepts

OOP is revolutionary. So it has a bunch of concepts for you to mug up.

2.1 Data Asbtraction

To show only the important parts to the outside world while keeping the implementations hidden. eg: *think about driving a car, you know how to but you probably can't build a car from scratch.*

2.2 Data Encapsulation

Means to keep data and methods that operate on some particular data together and restrict access to some of its components. eg: in Java we have private variables, getters and setters etc.

2.3 Inheritance

Inheritance is the property where one class obtains the methods and behaviors of another class. This topic is discussing more in-depth in Chapter 5.
eg:

```
class Animal  {}
class Dog extends Animal {}
```

2.4 Polymorphisms

Is derived from greek and means "many forms". Discussed more in detail in Chapter 4.

2.5 OOP vs Procedural Programming

Both are excellent paradigms for programming but have their key differences which makes them more useful for certain usecases.

Si.No	Procedural	Object Oriented
1.	The focus is on functions and procedures.	Focuses on objects which consist of data, behaviours and makes use of concepts such as abstraction, encapsulation, inheritance, polymorphism
2.	Data and functions are kept separate.	Objects are literally data + methods
3.	Good for small scripts.	Better for large applications
4.	Example: C, older Python etc.	Example: Java, C++ etc.

2.6 Microservices

A microservice in java is a software architecture where an application is split into small, independent services that communicate with each other via APIs (application programming interfaces).

Each service

1. Runs independently
2. Has its own database
3. Can be deployed independently
4. Does one task well.

So it prevents the whole program from failing because of one single component.

Chapter 3

Object Oriented Programming in Java

Java is inherently object oriented. It shoves it down your throat first thing when you write a java program. So it's important to know some stuff here and there.

3.1 Class Fundamentals

A class is the blueprint for an object. An object is just an instance of the class.

The general form of a class is:

```
class ClassName {
    type var1;
    type var2;

    type method(parameters) {
        ...
    }
}
```

As mentioned before, objects are created from classes.

```
// Declaring a reference
Car myCar;

// Creating the object
myCar = new Car();

// Both in one line
Car myCar = new Car();
```

The last line is what is known as **instantiation** and it means to declare a variable and initialize it in the same line.

3.2 Object reference

Now, the myCar in the example above does not hold any particular value. Rather it is a reference variable to that object i.e. think of a pointer in C/C++.

Which means, you can assign two variables to the same object via

```
Car c1 = new Car();
Car c2 = c1;
```

Now both c1 and c2 point to / reference the same object.

3.3 Methods

A method is a function inside of a class. Basically a normal function but in OOP we need to be as pretentious as possible and call it different names. eg:

```
class Car {
    void start() {
        System.out.println("Vroom vroom");
    }
}
```

You can invoke (call) these methods using the dot (.) operator on the reference variable i.e.

c1.start() will output Vroom vroom to the standard output.
Methods constitute the 'behaviour' part of an object whereas variables are just attributes.

3.4 Constructors

A constructor is a special method that is invoked during initialization of an object. It has

1. The same name as the class
2. No return type
3. Called automatically when you use 'new'

eg:

```
class Car {
    Car() {
        System.out.println("Car created");
    }
}
```

The above code ensures that every time you instantiate a Car object, it will print "Car created" to the standard output.

Constructors can also have parameters in them. This is useful if you know the properties of an object before creating it and can often be used in conjunction with method overloading (see Chapter 4).

3.5 Access Modifiers

Access modifiers determine what parts of the program can access what variables. There are 4 (technically 3) access modifiers in java.

1. default: free access within the same package only

   There is no keyword for default. All variables without any access modifiers are default by default.
2. public: free access everywhere
3. private: accessible only within the same class
4. protected: accessible from either the same class, or any subsequent subclasses within the same package.

These are mainly used to encapsulate data and is heavily used in places such as security systems etc.

3.6 The this Keyword

'*this*' in java is a keyword that is used to refer to the current object. If you've ever used python before, it's just like the 'self' keyword.

Some common uses of this are:

1. To differentiate between instance variables and parameters

```
class Car {
    String color;
    Car(String color) {
        this.color = color
    }
}
```

2. To call another constructor

```
class Car {
    this("Red");
}
```

3. To refer to the current object.

```
this.start()
```

Chapter 4

Polymorphism

As mentioned in the previous chapters, Polymorphism is Greek for "many forms". In java, that means we can have one method that can do multiple actions based on the information it receives.

For example: suppose we want to make an add method that can add either 2 or 3 integers or 2 doubles. We can do that in java like so

```
class MathAdd {
    int add(int a, int b) { return a+b; }
    int add(int a, int b, int c) { return a+b+c; }
    double add(double a, double b) { return a+b; }
}
```

Invoking the methods above, we can change the behaviour of the method by giving it different parameters.
Java picks the correct definition based on

1. Number of parameters
2. Type of parameters
3. Order of parameters

4.1 Objects as Parameters

You can pass objects as parameters to methods and even return them
eg:

```
// passing an object as a parameter
void updateCar(Car c) {
    c.color = "Red";
}

// returning an object
Car getCar(String color) {
    return new Car(color);
}
```

4.2 Recursion

A method calling itself is recursion. Very common across many types of data structures and algorithms

```
int factorial(int n) {
    if (n < 1) return 1;
    return n * factorial(n-1);
}
```

4.3 The static Keyword

Static is used to define a variable as a part of the class rather than as an attribute of an object. If a variable is declared as static, it can be access before any object of the class is created. eg: the main method is static.

```
class Counter {
    static int count = 0;
}
```

Which is accessed using Counter.count.

A whole block can be static

```
class UseStatic {
    static {
        System.out.println("Static block...");
    }
}
```

4.3.1 Restrictions

A static member has a few restrictions.

1. They can only call other static methods
2. They must access only static data
3. They cannot use ‘this’ or ‘super’ (Chapter 5) in any way.

4.4 The final keyword

Java doesn’t have const for some reason. This is the equivalent. Declaring a variable as final prevents it from being modifications in the future.

```
final int FILE_NEW = 1;
```

final variables don’t occupy memory on an instance basis.

1. Final variables cannot be changed once assigned
2. Final objects cannot reference other objects but its internal values can change
3. Final methods cannot be overriden
4. Final classes cannot be extended

4.5 Inner Classes

A class inside another class. Pretty simple definition, but there are different types of inner classes.

1. Member inner classes

```
class Outer {
    class Inner {
        void show() { ... }
    }
}
```

2. Static inner class

```
static class Inner { ... }
```

3. Local inner class: defined inside a method
4. Anonymous inner class: used when you need a one-time use class.

```
Runnable r = new Runnable () {
    public void run() {
        System.out.println("Running");
    }
}
```

Useful for event handling and simplifying code.

Chapter 5

Inheritance

Inheritance is the property by which a class inherits all the properties of another class.

The class with the properties is called the superclass and the one inheriting it is called the subclass.

At its most basic form, inheritance is like a parent-child relationship between classes.

To do inheritance, we use the extends keyword

```
class Animal { ... }
class Cat extends Animal { ... }
```

The subclass will now have:

1. variables
2. methods
3. NO constructors

of the superclass.

5.1 Types of Inheritance

Java supports 4 types of inheritances.

1. Single inheritance: one parent → one child

   ```
   class A {}
   class B extends A {}
   ```

2. Mutlilevel inheritance: a child inherits from a class that inherits from another class.

   ```
   class A {}
   class B extends A {}
   class C extends B {}
   ```

3. Hierarchical inheritance: one parent and multiple children.

```
class A {}
class B extends A {}
class C extends A {}
```

4. Mutliple inheritance: Java does not support multiple inheritances between classes. However, you can still do multiple inheritance using interfaces (Chapter 6)

```
class A implements interface_1, interface_2 ... {}
```

The reason Java doesn't support multiple inheritance is because of the diamond problem, ambiguity etc.

5.2 The super Keyword

Used to access superclass shi

```
// call parent constructor
super();

// access parent class variables
super.name;

// call methods from parent class
super.display();
```

5.2.1 Order of Constructors

```
class A {
    A() { System.out.println("A"); }
}

class B extends A {
    B() { System.out.println("B"); }
}

new B();   // prints A then B
```

Because Java ALWAYS builds the parent before the child.

5.3 Method Overriding

A subclass can override methods of its superclass.

```
class Animal {
    void sound() { System.out.println("Generic sound"); }
}

class Dog extends Animal {
    @Override
    void sound() { System.out.println("Bark"); }
}
```

5.4 Protected Members

protected members are accessible from

1. the same class
2. the same package
3. any subclasses (even from other packages)

5.5 Dynamic Method Dispatch

Also known as runtime polymorphism

```
Animal a = new Dog();
a.bark(); // from inside Dog class
```

Chapter 6

Packages And Interfaces

6.1 Packages

A package groups related classes together. At the top of every file in a package, include this line

```
package packageName;
```

Now your class belongs to packageName and file structure should match accordingly which looks like

```
packageName/
    MyClass.java
```

6.1.1 CLASSPATH

The CLASSPATH tells java where to look for classes and packages. You only need to mess with it if

1. Your classes are in custom folders
2. You're compiling/running from outside the project directory.

Example: (bash code so # is a comment)

```
set CLASSPATH=C:\myjava\packages;

# or

javac -cp . MyClass.java
java -cp . MyClass
```

Where . means current directory.

6.1.2 Access Protection With Packages

Packages affect the visibility of variables/methods.

Modifier	Same class	Same package	Subclass	Other packages
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	Yes	No	No	No

Default access = package-private

6.1.3 Importing Packages

You can import the package whole or just one class at a time.

```
import java.util.Scanner; // built in
import java.util.*; // all methods from util

// custom packages (user defined)
import myPack.*; // whole package
import myPack.MyClass; // single class
```

And then you can use the classes as normal for instantiation etc.

6.2 Interfaces

An interface is used to implement data abstraction in Java. It forces you to use total abstraction as it can contain only abstract methods and public static final variables. However, interfaces can be used to do multiple inheritance in Java.

An interface is defined using the 'interface' keyword

```
interface Shape {
    double area();
    void draw();
}
```

Notice how the interface doesnt have any implementations or even constructors?

6.2.1 Implementing an Interface

To implement an interface, use the 'implements' keyword
Example:

```
class Circle implements Shape {
    double r;

    Circle(double r) {
        this.r = r;
    }

    public double area() {
        return Math.PI * r * r;
    }

    public void draw() {
        System.out.println("Drawing circle");
    }
}
```

6.2.2 Accessing an Interface Implementation

To access implementations, we use interface references.

```
Shape s = new Circle(5);
System.out.println(s.area());
s.draw();
```

You can refer to any implementing class using the interface type.

6.2.3 Extending Interfaces

Interfaces can be extended just like classes

```
interface A {
    void show();
}

interface B {
    void display();
}

interface C extends A, B { // mulitple inheritance
    void hello();
}
```

Here, any class implementing C must implement all the methods of A and B also.

Chapter 7

Exception Handling

Exception is the java-lingo for errors. An exception happens when the program doesn't execute as expected, hence it is an exception.

7.1 Types of Exceptions

There are two broad categories of exceptions in java

1. **Checked exception**:
   Checked at compile time
   Examples: IOException, SQLException etc.
   The compiler forces you to deal with them
2. **Unchecked exception**:
   Occurs at runtime
   subclasses of RuntimeException
   Examples: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
   You don't need to explicitly handle them. They'll mess up your program without you ever accounting for them.

7.2 Handling Exceptions

Exceptions are dog shit, they need to be handled somehow. In java, we have the try-catch environments to help us

7.2.1 try and catch

Code prone to errors (exceptions) is put in the try block and if an exception does occur, the program will jump to the catch block

```
try {
    int x = 10 / 0; // raises an ArithmeticException
} catch (Exception e) {
```

```
    System.out.println("Division by zero error");
}
```

You can catch more than one exception although order does matter in this case

```
try {
    ...
} catch (Exception1 e1) {
    ...
} catch (Exception2 e2) {
    ...
} catch (Exception3 e3) {
    ...
}
```

Remember: Exceptions must go from Specific → General

If you put 'Exception' in the first catch block, the program will whine about it since it will always go to that catch block regardless of what error was raised.
Try-catch statements can also be nested

```
try {
    try {
        int a = 10 / 0;
    } catch (ArithmeticException e) {
        System.out.println("Inner error");
    }
} catch (Exception e) {
    System.out.println("Outer error");
}
```

And yes, this is awful practice so try to avoid it as much as possible.

7.2.2 The finally Keyword

Not to be confused with 'final', finally is used to execute a block of code regardless of whether an exception has occured or not

```
try {
    // risky code
} catch (Exception e) {
    // handle
} finally {
    System.out.println("Cleanup");
}
```

Typically used for cleanups. eg: If you opened some files before the try-catch block, you won't be able to close the file if an exception occurs which creates a dangling pointer and those are dangerous. Using finally solves this issue since it will run every time.

7.2.3 The throw and throws Keywords

'throw' is used to manually throw an exception

```
throw new IllegalArgumentException("Invalid input");
```

'throws' is used in method signatures to declare possible exceptions. It passes on the handling of exceptions to the person using it i.e. it makes the exception thrown checked. eg:

```
void readFile() throws IOException {
    ...
}
```

This means, every time readFile() is used, it will need to be wrapped in a try-catch where IOException is in at least one of the catches.

7.3 Custom Exceptions

You can make your own exceptions. How cool is that?

To make one, extend the Exception or RuntimeException class.

```
class MyException extends Exception {
    MyException(String msg) {
        super(msg);
    }
}
```

and then, elsewhere, you can throw it

```
throw new MyException("Custom error caught!");
```

7.4 Design Patterns in Java - Singleton

This pattern ensures that only one instance of a class exists in the entire application. It is used when only one object is needed for the whole program. Used in:

1. Loggers
2. Configuration managers
3. Database connections

A basic implementation is given below

```
class Singleton {
    private static Singleton instance;

    private Singleton() {
        // private so no one can create objects
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

This has the problem of not being thread safe however and can mess up in multi-threaded environments. To fix that, we can add the synchronized keyword

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

You can choose to initialize this one of two ways

1. Eager initialization: Keep it initialized from the beginning (sometimes better for multithreaded environments)
2. Lazy initialization: Initialize only when needed.

7.5 Design Patterns in Java - Adapter

Used to implement legacy methods into modern systems i.e. turn something incompatible into something compatible.

Use this design pattern when you need to use an existing class but its interface doesn't match what your code needs.

It consists of 4 components

1. **Target interface**: The set of operations that the client code wants to use.
2. **Adaptee**: the old system that has the incompatible interface and needs to be converted.
3. **Adapter**: A class that implements the target interface and uses and instance of the adaptee to make it compatible with the target interface.

   It acts as a bridge, adapting the interface of the adaptee to match the target interface.
4. **Client**: The code that uses the target interface to interact with objects. It is completely unaware of the shit regarding adaptee-adapter implementations.

   This is the code that benefits from the integration of adaptee into the system via the adapter.

Example: Suppose you have round pegs and square pegs. You can't insert a round peg into a square hole, so a conversion is necessary.

```
// Adaptee: Existing incompatible class
class RoundPeg {
    public void insertIntoRoundHole() {
        System.out.println("Round peg in round hole");
    }
}

// target interface
interface SquarePeg {
    void insertIntoSquareHole();
}

// Adapter: makes the RoundPeg behave like a SquarePeg
class PegAdapter implements SquarePeg {
    private RoundPeg roundPeg;

    PegAdapter(RoundPeg rp) {
        this.roundPeg = rp;
    }

    public void insertIntoSquareHole() {
        // convert call
        roundPeg.insertIntoRoundHole();
    }
}
```

The methods from PegAdapter can be used by some other class to do appropriate functions now.

Usage:

```
RoundPeg rp = new RoundPeg();
SquarePeg sp = new PegAdapter(rp);
sp.insertIntoSquareHole(); // this will work
```

Chapter 8

SOLID Principles

SOLID principles are 5 principles that help you to write better OOP code. It isn't restricted to only java and you can use these for any object oriented language

8.1 Single Responsibility

For every class in your program, one class should be responsible for one task only.

eg: Given below is a bad example because UserService doesn't need to handle sending emails as well.

```
class UserService {
    void addUser() {}
    void sendEmail() {}
}
```

To fix this,

```
class UserService { void addUser() {} }
class EmailService { void sendEmail() {} }
```

8.2 Open/Close Principle

All classes should be open to extension but closed to modification i.e. build your classes in a way that new features can be added on and the existing class doesn't need to be edited from the inside

eg: Use inheritance or interfaces instead of rewriting logic

8.3 Liskov Substitution

All subclasses should be interchangeable with their superclasses i.e.
If class B extends A, then objects of A can be swapped with B safely.

Some **negative** examples are:

1. A subclass that removes some functionality
2. Or throws some weird exceptions

8.4 Interface Segregation

Split larger interfaces into smaller ones wherever possible.

eg: Consider the interface below

```
interface Worker {
    void work();
    void eat();
}
```

If we decide to employ robots in the future, them eating wouldn't make much sense (maybe they consume fuel but idk man that's not really 'eating' yk?)

A better alternative to this would be:

```
interface Workable { void work(); }
interface Eatable { void eat(); }
interface Fuelable { void refuel(); }
```

And we can use multiple inheritances to make the necessary changes.

8.5 Dependency Inversion

Depend on abstractions instead of concrete implementations wherever possible.

That means higher level modules should be dependent on interface and not the specific implementations.

eg: bad example

```
class Service {
    ConcreteLogger logger = new ConcreteLogger();
}
```

A better alternative

```
class Service {
    Logger logger;
    Service(Logger logger) { this.logger = logger; }
}
```

Makes testing easier.

Makes swapping implementations easier.

Makes *life* easier.

Chapter 9

Swing Fundamentals

In the past, java developers used something called the Abstract Window Toolkit (AWT) to create graphical user interfaces (GUIs).

However, this was not with its set of problems. First of all, AWT applications all look ugly and coding in java was already hell without IDEs back then, so you can imagine how much worse it would be to create a full blown application. Furthermore, the applications were OS dependent, heavier, and overall had a very limited look and feel to it.

To combat these, java introduced the swing package as a part of its extended packages. It is built entirely on top of AWT and still written entirely in Java.

However, it is not nearly as insufferable as AWT and has the added features of: lightweight components, supports pluggable look and feel, more powerful and flexible, faster to develop. In other words, If AWT had a girlfriend, Swing is the guy she told him not to worry about.

To use swing, you'll need to import it from the extended packages

```
import javax.swing.*;
```

9.0.1 Some Key Features of Swing

1. Lightweight components
2. Pluggable look and feel
3. Model View Controller architecture
4. Double buffering
5. Rich set of widgets
6. Highly customizable.

9.1 Model View Controller (MVC) in Swing

A visual component is a composite of three distinct aspects:

1. The way the component looks.

2. The way it reacts to the user.
3. The state information associated with the component.

The **model** corresponds to the state information associated with a component.
The **view** determines how the component is displayed on the screen.
The **controller** determines how the component reacts to the user.

Swing combines the view and controller into a single logical entity (UI delegate).This approach is called the Model-Delegate architecture or Separable Model architecture.

The model is responsible for maintaining information about the component's state.

The UI delegate is responsible for maintaining information about how to draw the component on the screen.
It has the ability to tie multiple views to a single model

9.2 Components and Containers

A component is an independent UI element eg: button, label, inputfield etc.

A container is used to hold a group of components or in other words, a container is a special type of component that can hold other components eg: frame, panel, dialog etc.

9.3 Swing packages

As mentioned before, to use swing we'll need to import it from javax. Some other packages include

```
import javax.swing.*; // general purpose
import javax.swing.event.*; // for event handling
import javax.swing.border.*; // for borders
import javax.swing.table.*; // for table models
```

9.3.1 Event Handling in Swing

Event driven programming is done via the use of listeners and adapters. More will be discussed about this in Chapter 10
eg:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
});
```

9.4 Layout Managers

Swing contains layout managers that determine how components are placed inside a container.

Some common ones are:

1. FlowLayout() - one line left to right
2. BorderLayout() - North, south, east, west, center
3. GridLayout() - rows × columns
4. BoxLayout() - vertical/horizontal stacking
5. GridBagLayout() - a much more flexible GridLayout() but also a lot more annoying to use

9.5 Some Core Swing Components

We'll be looking at the following:

1. JFrame
2. JLabel
3. JButton, JRadioButton, JCheckbox
4. JTextField

9.5.1 JFrame

The main window of the GUI

```
JFrame f = new JFrame("My Frame");
f.setSize(400, 400);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
f.setVisible(true);
```

9.5.2 JLabel

Just a basic label used to display text

```
JLabel label = new JLabel("Hello!");
```

9.5.3 JButton

You can click it

```
JButton b = new JButton("Click me");
```

9.5.4 JTextField

Used to input text

```
JTextField tf = new JTextField(20);
```

Chapter 10

Event Handling in Java

To handle events such as button presses, mouse movement, closing windows etc. we need to use event driven programming.

When an event occurs

1. An event object is created
2. It's sent to the corresponding event listener
3. Listener executes the code to handle it

10.1 Delegation Event Model

This is the backbone of event handling in java. It has three parts to it

1. Event source: the component that generates the event.
2. Event object: information about the event .
3. Event listener: interface with methods to handle the event.

The source delegates the event to the listener.

10.1.1 Event Classes

```
java.awt.event.*
javax.swing.event.*
```

Common event classes:

1. ActionEvent – button clicks
2. MouseEvent – mouse interactions
3. KeyEvent – keyboard
4. WindowEvent – window actions
5. ItemEvent – item selection
6. TextEvent – text changes

These classes store event-specific info (coordinates, which key, etc.)

Any UI component can be a source. Event sources are components that can fire events.

10.2 Event Listener Interfaces

Listeners define callback methods that run when an event occurs.

10.2.1 ActionListener

```
void actionPerformed(ActionEvent e);
```

10.2.2 MouseListener

```
mouseClicked(MouseEvent e)
mousePressed(MouseEvent e)
mouseReleased(MouseEvent e)
mouseEntered(MouseEvent e)
mouseExited(MouseEvent e)
```

10.2.3 MouseMotionListener

```
mouseDragged(MouseEvent e)
mouseMoved(MouseEvent e)
```

10.2.4 KeyListener

```
keyPressed(KeyEvent e)
keyReleased(KeyEvent e)
keyTyped(KeyEvent e)
```

10.3 An Example Using The Delegation Event Model

```
import javax.swing.*;
import java.awt.event.*;

class Demo {
    public static void main(String[] args) {
        JFrame f = new JFrame("Demo");
        JButton b = new JButton("Click");

        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        });

        f.add(b);
        f.setSize(300,200);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Chapter 11

Database Applications Using JDBC

Java Database Connectivity (JDBC) is an API used to connect java applications to some relational database like MySQL, PostgreSQL etc.

It lets you

1. Connect to a Database
2. Run SQL queries
3. Update data
4. Retrieve items

Basically, it acts as the middleman between the program and the database.

11.0.1 Types of Drivers

1. JDBC-ODBC: uses ODBC under the hood. It is slow and obsolete.
2. Native API: Uses some platform-specific native libraries.
   It is faster, but not portable.
3. Network Protocol: Uses some middleware server and thus provides better portability.
4. Thin Driver (pure Java): Directly talks to the database.
   Fast and portable
   Most commonly used today
   eg: MySQL connector etc.

The thin driver is the only one that really matters for us. The rest are meh

11.1 Steps Involved in JDBC

There are 5 classic steps:

Step 1: Load the driver

Modern java does this automatically, but prior to JDBC 4, we had to manually include this line

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Step 2: Establish a connection

```
Connection con = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/mydb",
"user",
"password"
);
```

Step 3: Create statement

Used to send some SQL queries

```
Statement stmt = con.createStatement();
```

Step 4: Execute query

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```

Step 5: Close connection

```
con.close();
```

11.2 Common JDBC Components

1. DriverManager: manages database drivers and gets connections
2. Connection: represents active connection to the database
3. Statement: Used to send SQL commands.
4. ResultSet: Holds data returned by SELECT queries
5. SQLException: Handles database-related errors.

11.3 A Connection Example

```
import java.sql.*;

class Demo {
    public static void main(String[] args) {
        try {
            // Step 1 (optional in modern Java)
            Class.forName("com.mysql.cj.jdbc.Driver");
```

```
            // Step 2
            Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydb",
            "root",
            "password"
            );

            // Step 3
            Statement stmt = con.createStatement();

            // Step 4
            ResultSet rs = stmt.executeQuery("SELECT * FROM student");

            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " +
                ↪  rs.getString(2));
            }

            // Step 5
            con.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```