OOP Quick Notes

Brief Intro to OOP

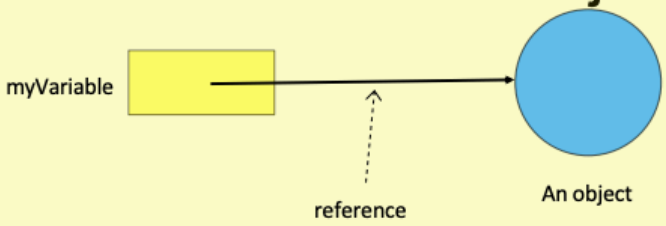
- using Java
- functions are **methods** stored in a class (has to be the same name as the filename)
- strongly typed language (must declare the type of variable before declaring the name of the variable, and you cannot change the type of the variable)
- no direct access to memory (as Java code is run in JVM - Java Virtual Machine), so references are used
- Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and colour, and **methods**, such as drive and brake.
- classes are capitalised
- objects are created from classes, e.g.

```
// Create an object called "`myObj`" and print the value of x:
public class Main {
  int x = 5;

    public void myMethod(){
        System.out.println("Hello world!");
    }

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

- All objects are accessed via object references:



- Objects exist in *heap* memory only.
  - Like managed C dynamic memory allocation.
  - But garbage collection automatically frees memory.

    MyClass myVariable = new MyClass();
  - Create and initialise a MyClass object, and obtain a reference to it.
  - myVariable must be declared as type MyClass.

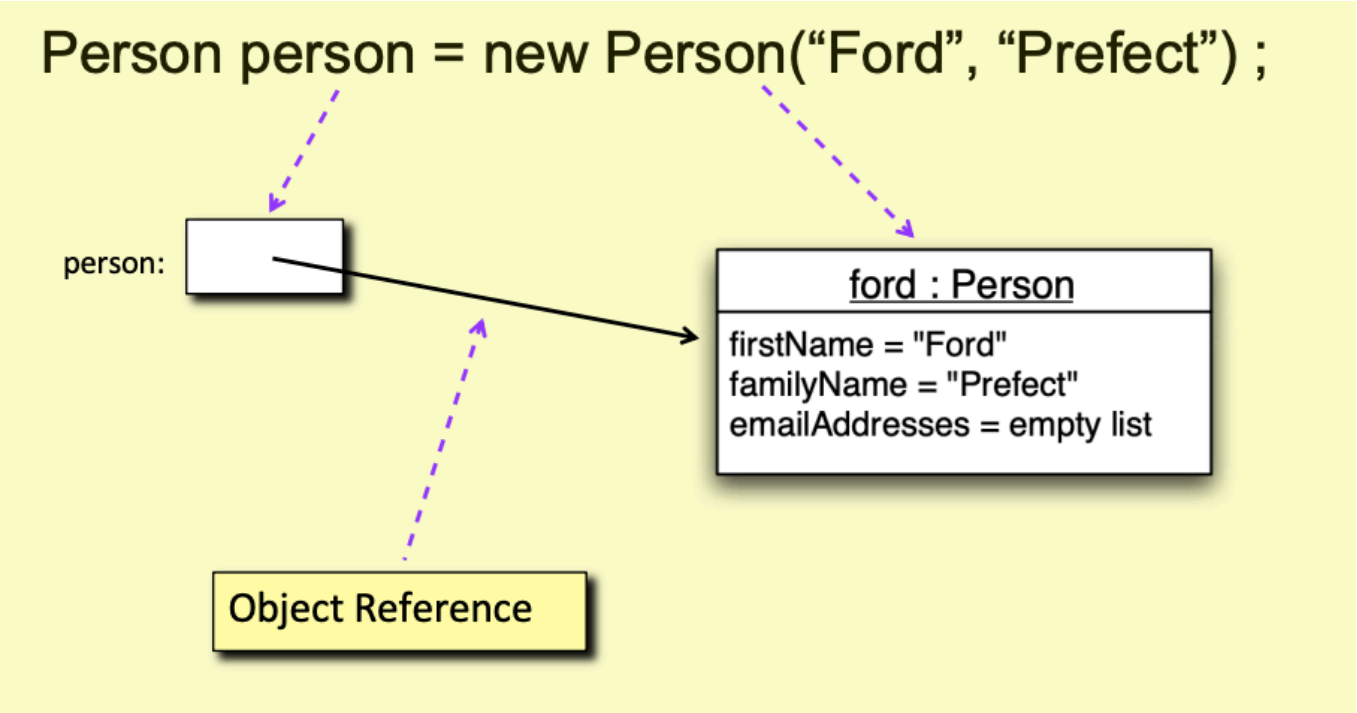
- You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).

```
// another class which craetes a Main object – but accessed in Second class
class Second {
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x); // printing out attribute x of myObj object
    myObj.myMethod(); // calling a method from myObj object (from Main
class)
  }
}
```

Note:
classes are represented by

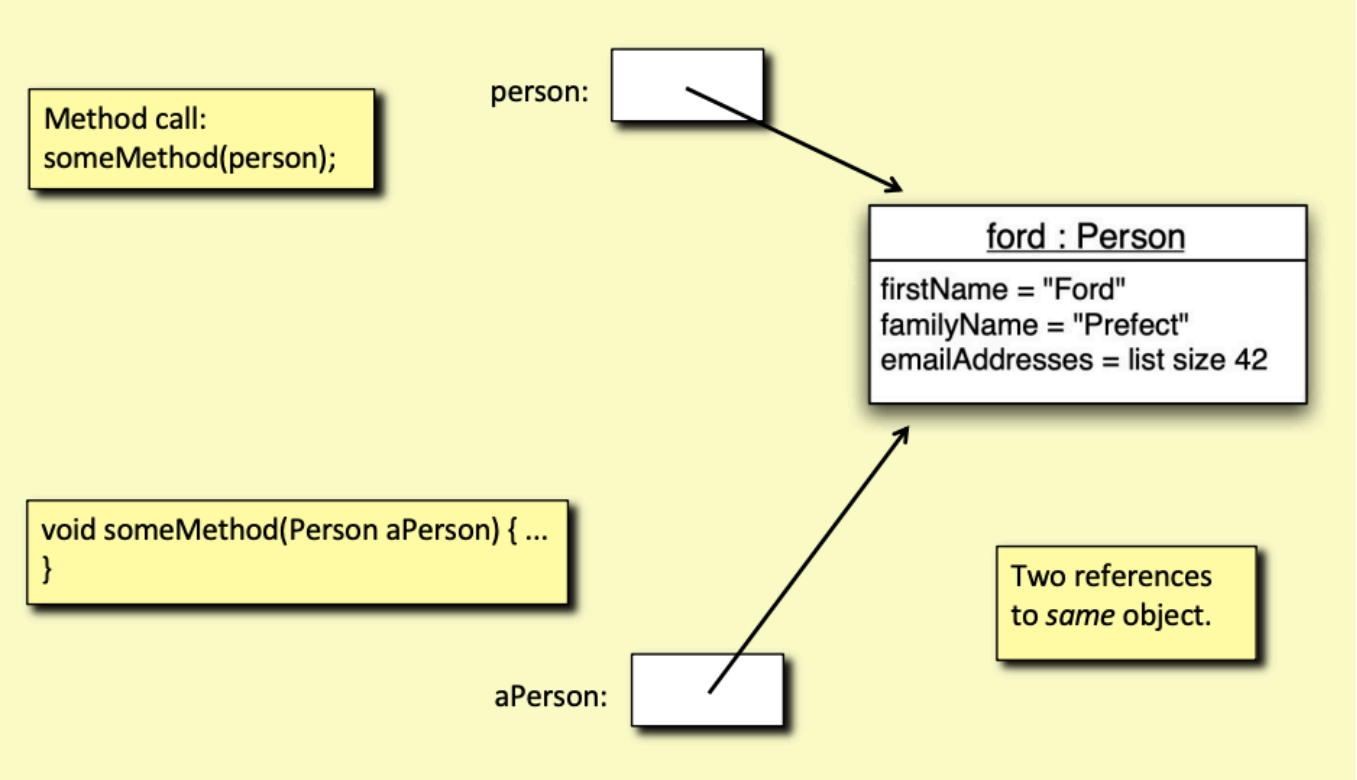
- Classes can be represented by:
  - Bytecode
    - produced by the compiler (.class files).
    - not (easily) human readable.
  - Source code
    - the programming language you use and write programs with.
  - A Modelling language
    - formal: UML
    - informal: notes, diagrams, doodles

object referencing



- creating a variable with the class type gives you the reference to the object initialised (you store the reference to the object in the variable, not the object itself)

calling a method on a reference (note that objects are not copied over)



- If an object reference is passed to a method:
  - Changing the object inside the method changes the object outside the method.
  - They are the same object!
- Don't forget that arrays are objects.
  - An array variable holds a reference to an array object.
  - Changing elements in an array passed to a method changes the array outside the method.

Public vs Private

private:

- A class defines a scope.
- Declaring a method or variable private means that it can be accessed only within the scope of the class.
  - This means within a method body or an instance variable initialisation expression.

- At runtime objects implement the scope rules.
  - Class + compiler + type checking ensures behaviour must conform.
- The internal state of an object should be private and changed by the object's methods only.
  - The state is represented by the values of the instance variables

public:

- Methods and variables declared public can be accessed by anything that has a reference to an object of the class.
  - Variables belong to an object.
  - Methods are called on an object.
- Methods form the public interface of objects of the class.
  - The services the object can perform.

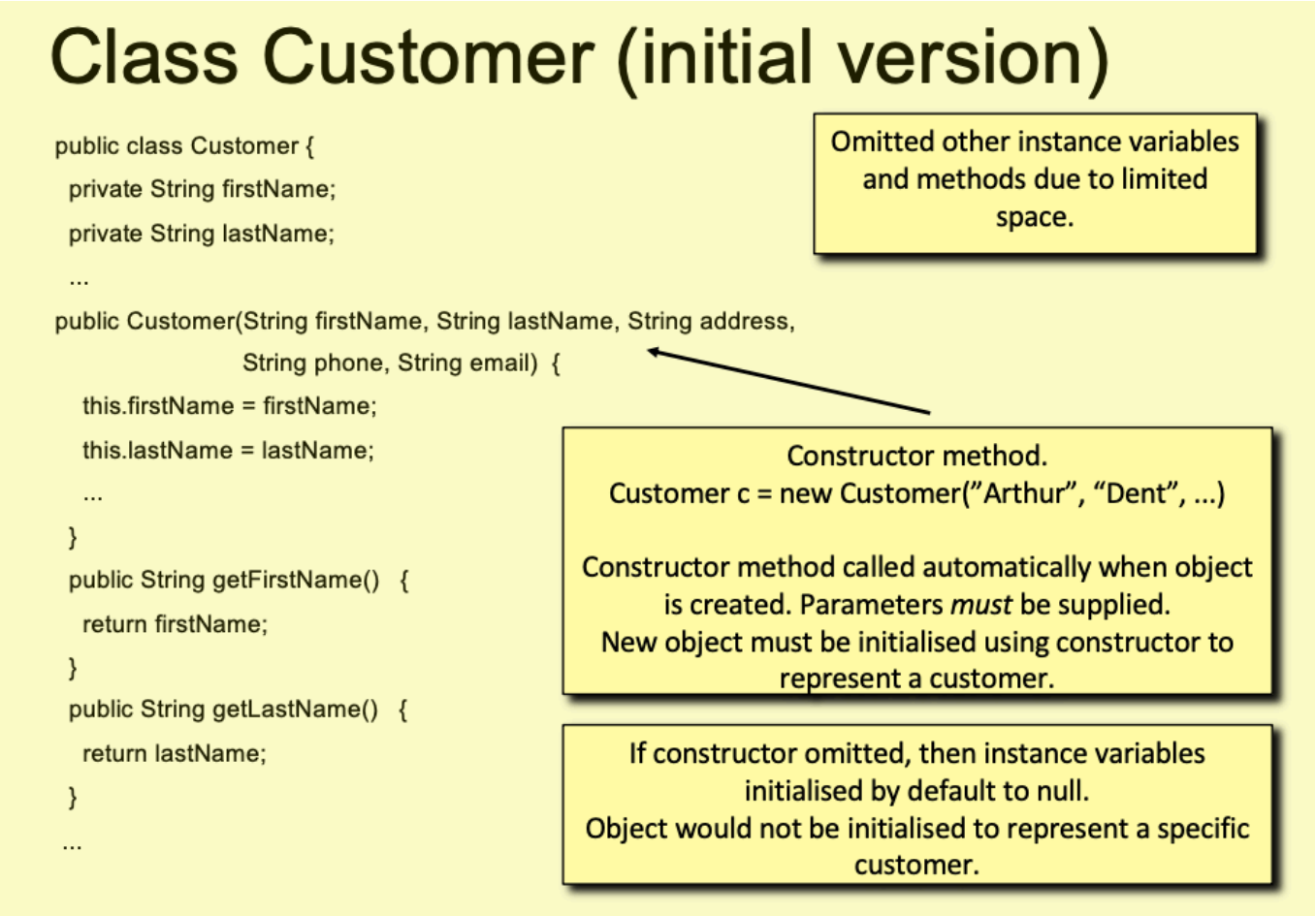
Class Constructors

initialise the attributes for the objects when initialised at runtime or intialised during running of program

this keyword used as a reference to the stored attributes, (**this** is used to refer to instance varaibles -> variables initialised at runtime, which are initialised by constructor)

```
public class Car{
    private String make;
    private int year;

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

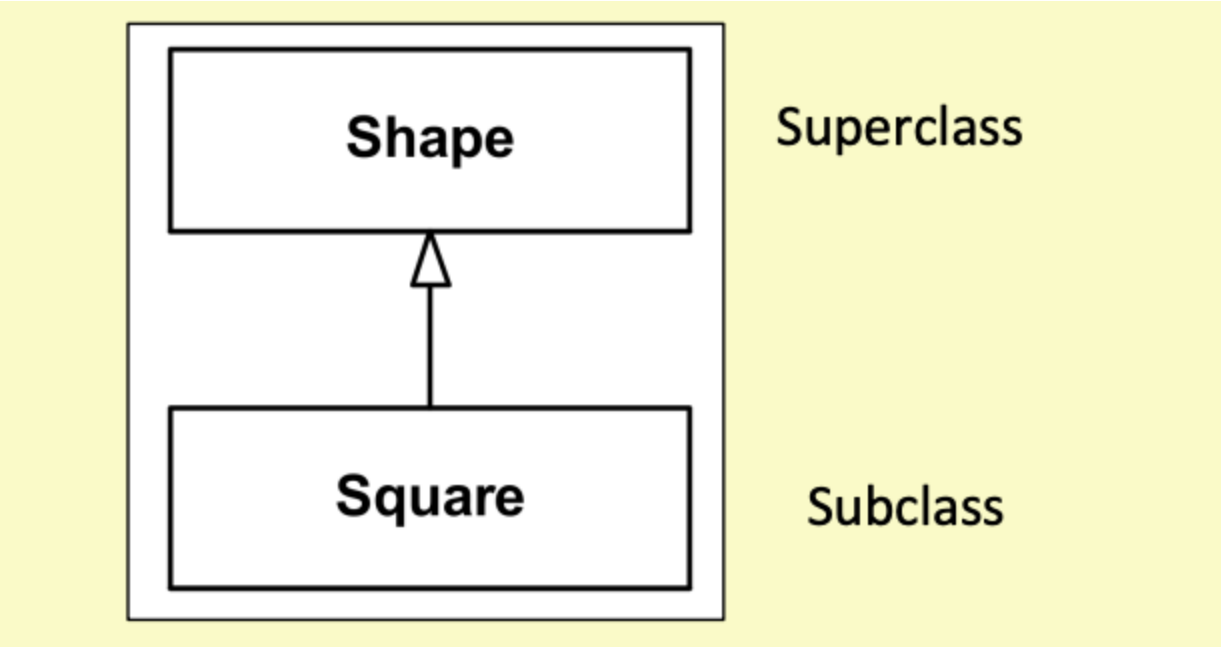


Encapsualtion (getters/setters)

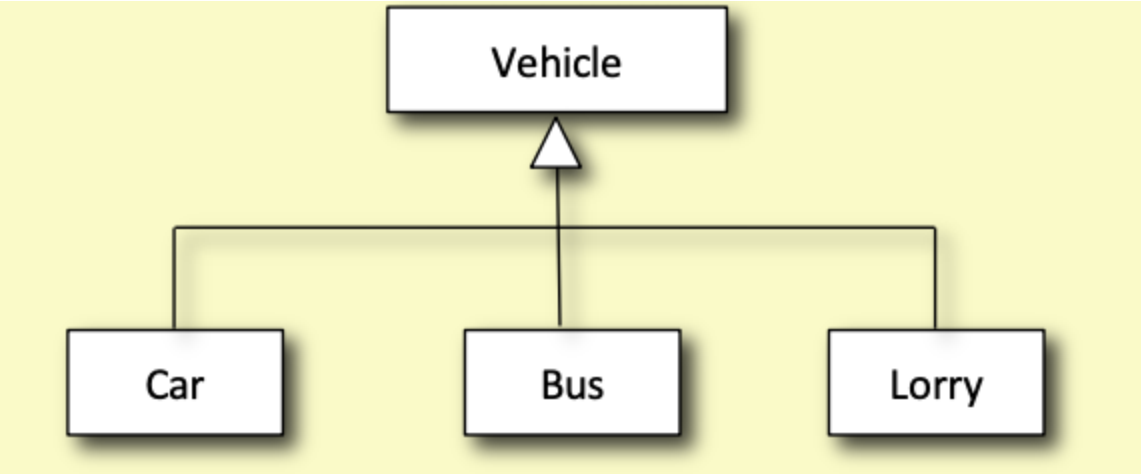
Inheritance

subclass vs superclass

- A subclass inherits from a superclass.
- The subclass gains all the properties of the superclass, can specialise, it and can add more features.



- more than 1 subclass can inherit from a single superclass



inheritance code example

```
public class Square extends Shape
{
   private int size ; // Need a size
   public Square(int x, int y, int size)
   { // To be added }
   public void draw(Graphics g)
   { // To be added }
   public void move(int x, int y)
   { // To be added }
}
```

New Keyword

- note that protected variables can be accessed from subclasses, and that private variables are inherited and part of subclass (BUT only can be accessed by superclass methods)

subclass constructor

```
public Square(int x, int y, int size)
{
   super(x,y) ; // Another new keyword
   this.size = size;
}
```

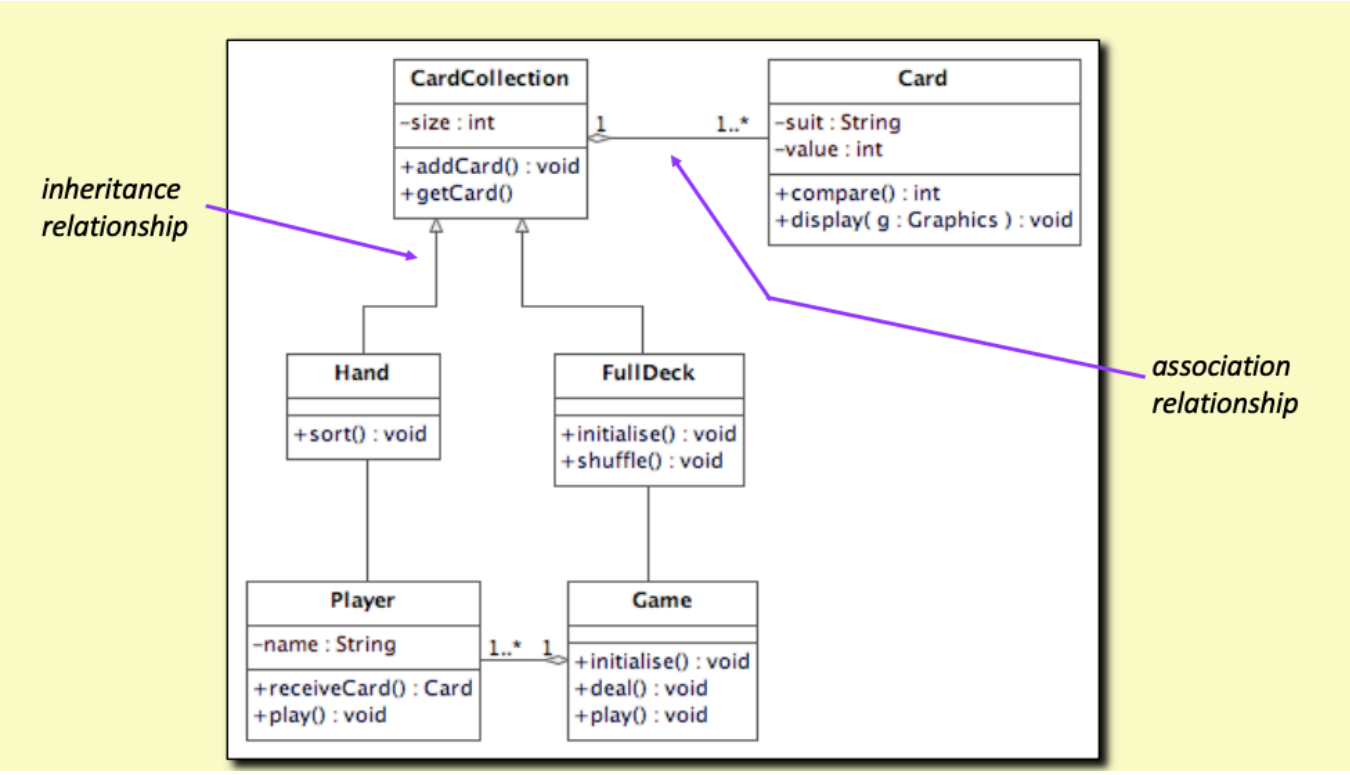
- *super* gives a way of referring to the superclass.
- When used in a constructor like this, it results in a call to the superclass constructor with the matching parameter list.
  - Known as 'explicit constructor invocation'.

- super **must** go first (it must be the first statement in the constructor body)

Abstract Classes

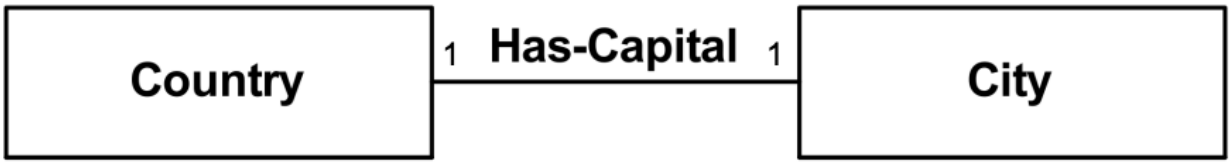
- Declaring an abstract method also forces the class to be declared abstract.
  public abstract class Shape {
  ...
  }
- An abstract class can have no instances.
- It is a partial description that can be extended.
- method declared without a body in abstract class (the method must be implemented by a subclass that inherits from the abstract class)
- Provides “shared via inheritance” instance variables/methods.

UML Diagram



- A program consists of a collection of classes.
- Those classes define the abstract structure of the program in terms of the relationships or associations between the classes.
- When the program is run, the associations are realised by object references (links in UML).

examples:



- This represents the abstraction of:
  - A country having a capital city.
  - A city being the capital of a country.
- At any one point in time a Country object will be linked to one City object.
- Many Country and City objects may exist but can only be linked one-to-one.
- The association Has-Capital is bi-directional.

- In Java an association between two classes is usually realised by an object of one the classes having a reference to an object of the other class.

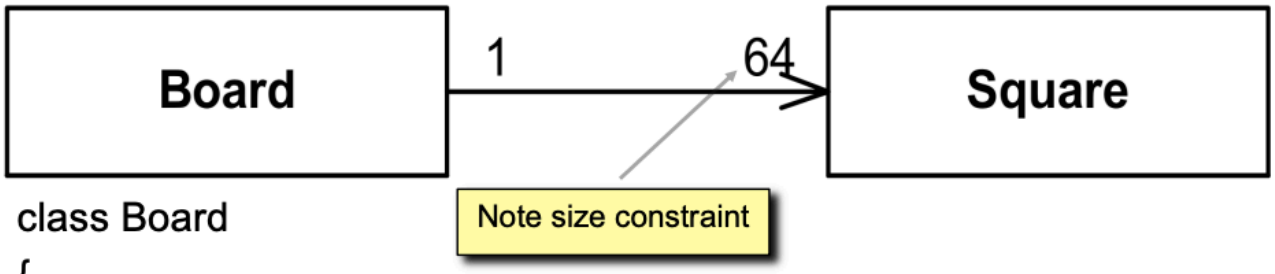
```
class Country
{
        private City capital;
        ...
}
```

In UML an association is bi-directional by default.
In the implementation, we usually want a uni-directional reference only. This can be made explicit in UML by adding an arrow head to one end of the association.

Has-Capital is now a *navigable* association (uni-directional).

example 2:

- The type used to represent the association needs to be determined correctly.

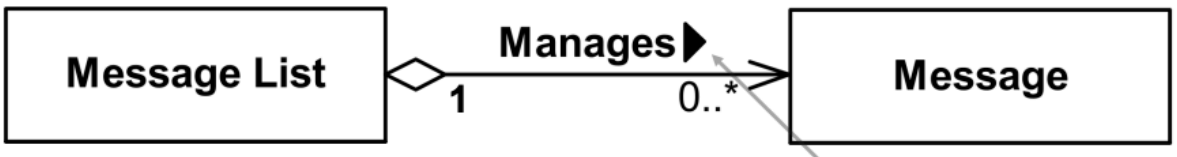


Note size constraint

```
class Board
{
  private Square[] squares = new Square[64];
  // or
  // private ArrayList<Square> squares = new ArrayList<Square>();
  ...
}
```

example 3:

- Aggregation associations are realised in the same way but we may need to be careful about sharing objects.

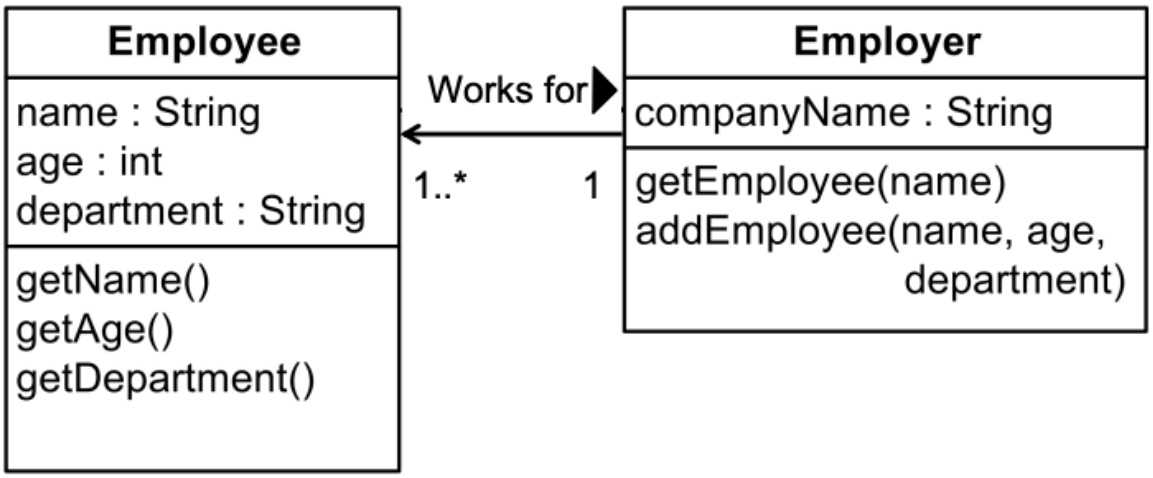


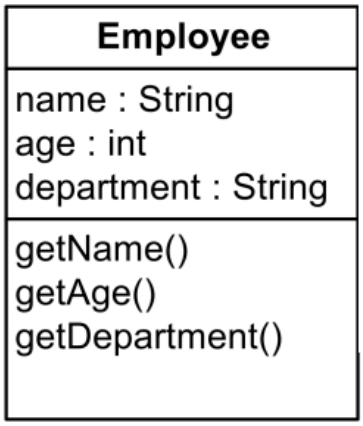

```
class MessageList
{
  private List<Message> list = new ArrayList<Message>();
  ...
}
```

An aggregation allows Messages to be in several lists.

The triangle annotation is a way of showing which direction the Manages relationship applies to.

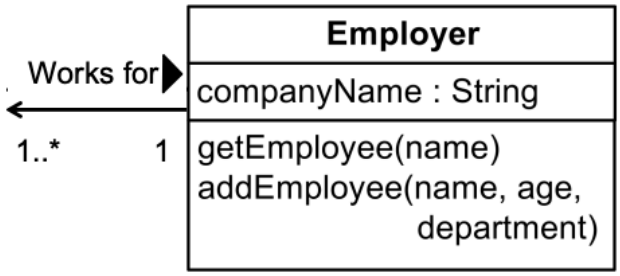
example 4:






```
class Employee {
   private String name;
   private int age;
   private String department;

   public Employee(String name, int age,
                           String department) {
      this.name = name;
      this.age = age;
      this.department = department;
   }
   public String getName() { return name; }
   public int getAge() { return age; }
   public String getDepartment()
   { return department; }
}
```

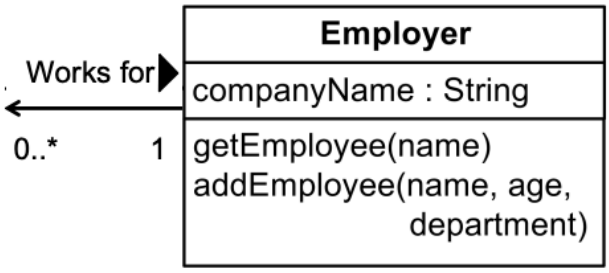



```
class Employer {
   private String companyName;
   private ArrayList<Employee> employees;

public Employer(String companyName) {
  this.companyName = companyName;
  employees = new ArrayList<Employee>();
}
```

```
public Employee getEmployee(String name) {
  for (Employee employee : employees) {
    if (employee.getName().equals(name))
    { return employe; }
  }
   return null;
}

public void addEmployee(String name,
                        int age,
                        String department) {
 Employee employee =
   new Employee(name,age,department);
 employees.add(employee);
}
}
```




```
class Employer {
   private String companyName;
   private ArrayList<Employee> employees;

public Employer(String companyName) {
  this.companyName = companyName;
  employees = new ArrayList<Employee>();
}
```

```
public Employee getEmployee(String name) {
  for (Employee employee : employees) {
    if (employee.getName().equals(name))
    { return employee; }
  }
   return null;
}

public void addEmployee(String name,
                        int age,
                        String department) {
 Employee employee =
   new Employee(name,age,department);
 employees.add(employee);
}
}
```

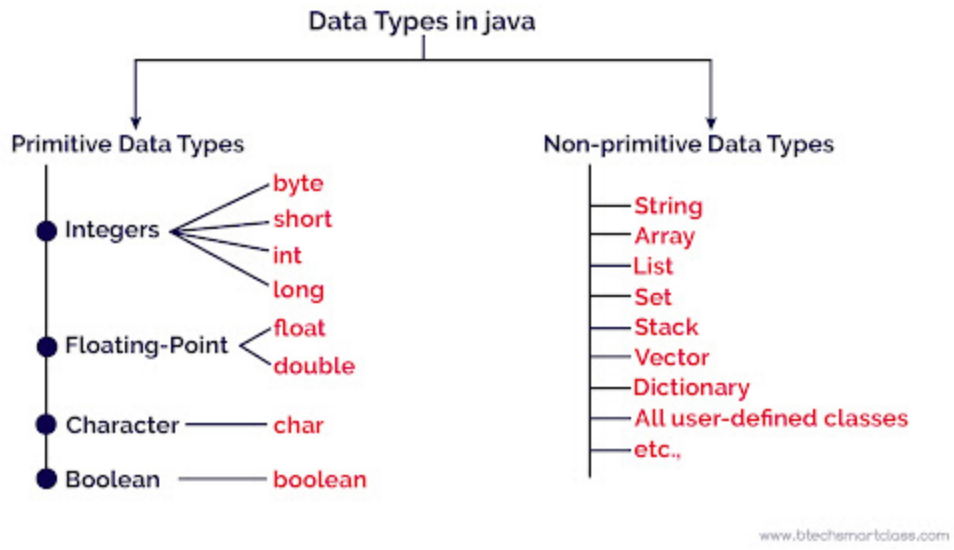
Method Overloading

- A class can have more than 1 constructor
- Each can be used to initialise objects in a specific way
- BUT the constructor must have the same name, i.e. same name as the class

- overloading: when 2 or more methods/constructors have the same name but different parameters
- note that they have to return the same data type
- compiler determines which method to call by matching the argument types

Java Syntax

Data types:



primitive data types:

Type	Description	Default	Size	Example Literals
boolean	true or false	false	1 bit	`true, false`
byte	twos complement integer	0	8 bits	(none)
char	Unicode character	`\u0000`	16 bits	`'a', '\u0041', '\101', '\\', '\'', '\n', 'ß'`
short	twos complement integer	0	16 bits	(none)
int	twos complement integer	0	32 bits	`-2, -1, 0, 1, 2`
long	twos complement integer	0	64 bits	`-2L, -1L, 0L, 1L, 2L`
float	IEEE 754 floating point	0.0	32 bits	`1.23e100f, -1.23e-100f, .3f, 3.14F`
double	IEEE 754 floating point	0.0	64 bits	`1.23456e300d, -1.23456e-300d, 1e1d`

- primitive data types -> values stored direclty in variables
- class types (non-primitive data types) -> values stored in variables are references to objects in heap (note String is not a primtiive data type.)

check string equal:

```
String s1, s2;
s1.equals(s2);
```

- equals is used to compare the contents of strings
- == is used to compare the references of the string (whether are they the same String objects). That is why == is used in primitive data types (String is not an example of one)

get length of a string

```
String word = "worm";
System.out.println(word.length()); // 4
```

get index of first occurrence of character in string

```
String fruit = "apple";
System.out.println(fruit.indexOf("a")); // 0
```

convert string to uppercase

```
String team = "denver nuggets";
String uppercasedTeam = team.toUpperCase(); // "DENVER NUGGETS"
```

convert stirng to character array

```
String team = "apple";
char[] myCharArr = team.toCharArray();
```

split a string by delimitter

```
String player = "Cristiano,Ronaldo,7,6 ft 1, Al Nassr"
String[] split_string = String.split(",");
print(split_string) // ['Cristiano","Ronaldo",...]
```

iterate over String

```
String name = "Messi";
for(int i = 0; i < name.length(); i++){
    char c = name.charAt(i);
    System.out.println(c);
}
```

Conditionals

```
if(condition){

}
else if(condition){

}
else{
```

```
}
```

ternary operator (shortcut of if..else) -> with 3 operands - (**condition** ? true : otherwise):

```
int age = 18;
// if condition true -> follow ?, otherwise follow :
boolean tooOld = (age >= 18) ? true : false
```

switch

comparing
In Java, using the == operator to compare objects actually compares the object references. The operator will return true if the two references are the same, meaning that both references refer to the same object. Using == to compare two references to different objects will always return false as the references must be different even if the two objects represent the same value.

To compare strings properly you need to use the equals and compareTo methods. The equals method will return true if two strings are identical (that is they contain exactly the same characters in the same order). String s1 = ... String s2 = ... boolean b = s1.equals(s2); b is initialised to true if the strings are the same, false otherwise. You can also use the method equalsIgnoreCase, which will ignore the difference in case between letters, so that A is considered the same as a and so on.

The compareTo method provides the equivalent of ==, < and >.

```
String s1 = ...
String s2 = ...
int n = s1.compareTo(s2);
```

compareTo will return a value less than zero if s1 comes before s2 alphabetically (for example "hello" comes before "world" in dictionary order). compareTo will return a value of zero if s1 is the same as s2. compareTo will return a value greater than zero if s1 comes after s2 alphabetically

Looping

Conditionals:

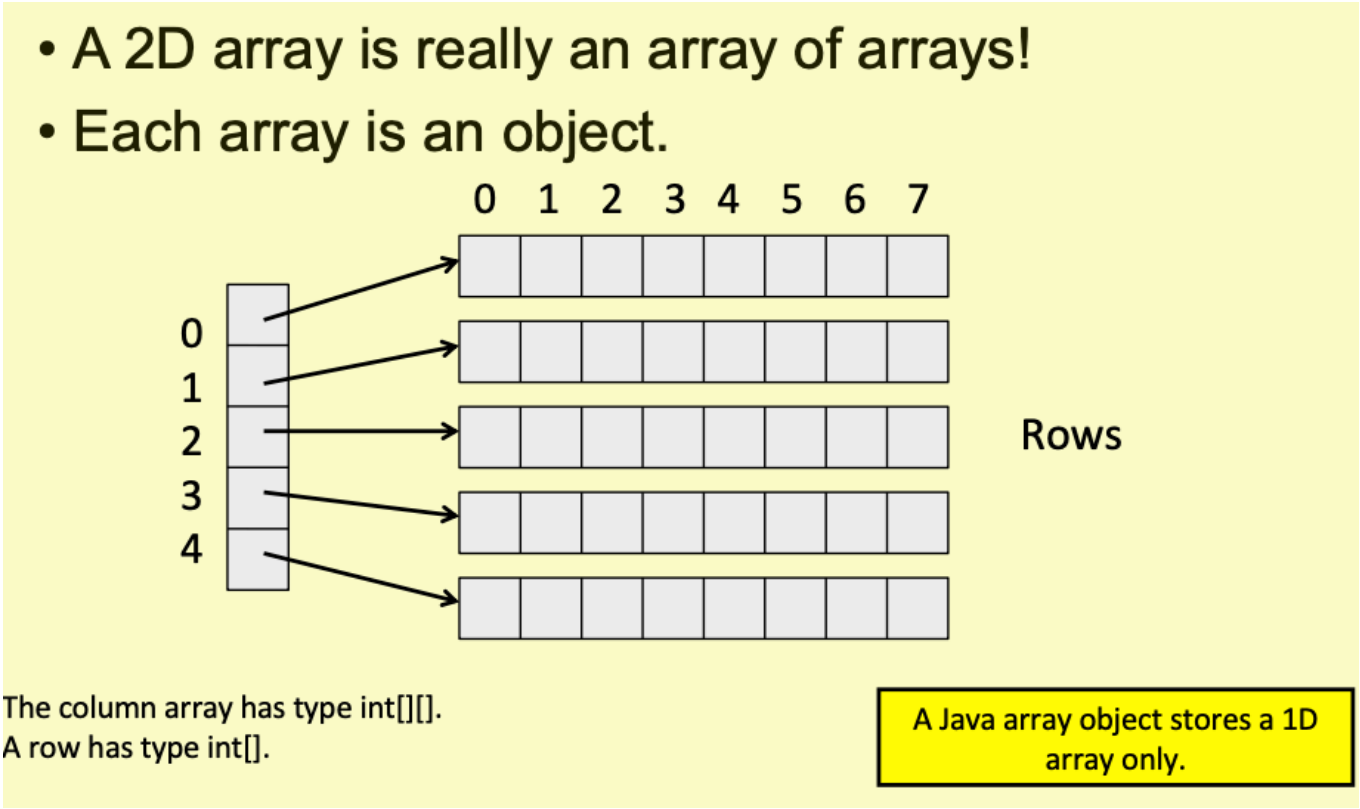
Arrays

Hold fixed size collections

length of array is known via the .length attribute (as ararys are objects) -> arrays in Java are managed, compared to C (which is just memory locations)

```
a.length // return size of array
```

2d arrays:



ArrayList

Java Records

- A kind of class specialised for representing data objects.
- Where an object is used as a collection of fields (instance variables), like a C struct.
- Doesn’t need additional methods but they can be added.
- Objects are shallowly immutable.
  - Instance variables final but referenced objects can change, ArrayList of values.
- Provides a compact syntax for declaring record classes.

example:

```
public record Book (String title, String author, int pages) {}
```

- The fields (type, name) are listed in parentheses.
- Nothing in the braces.
- Objects created in usual way:
  - Book book = new Book("title", "name", 200);
- includes boilerplate code e.g. toString, getters, setters, equals method to compare object values, which are generated automatically
- getter methopd added with same name for each field, without the "get'Name'" e.g.
  - String book = book.title();
  - int pages = book.pages();
- fields are final, as instance variables cannot be changed by assignment
- can add methods to a record e.g.

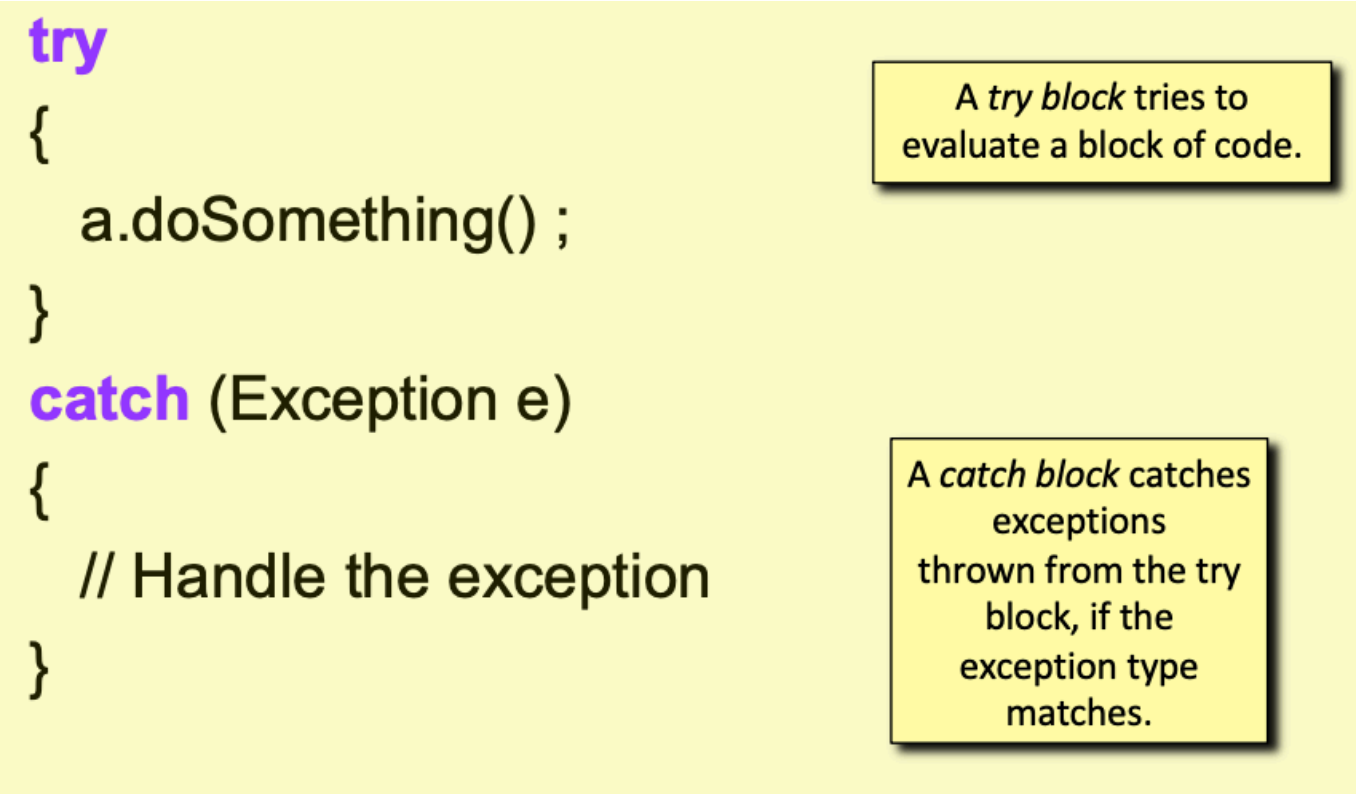
Exceptions

Throwing exceptions

```
public T pop() throws EmptyStackException
{
    if (values.size() == 0) // empty
    { throw new EmptyStackException(); } // Note no return needed here
    else
    { return values.remove(getLastIndex()); }
}
```

- Make the program deal with the error.
- Make the compiler check the code.
- Make the programmer write the code properly!
  - Use the stack in the way it is designed.
- Allow recovery to be made.
- Recognise that pop cannot return normally.

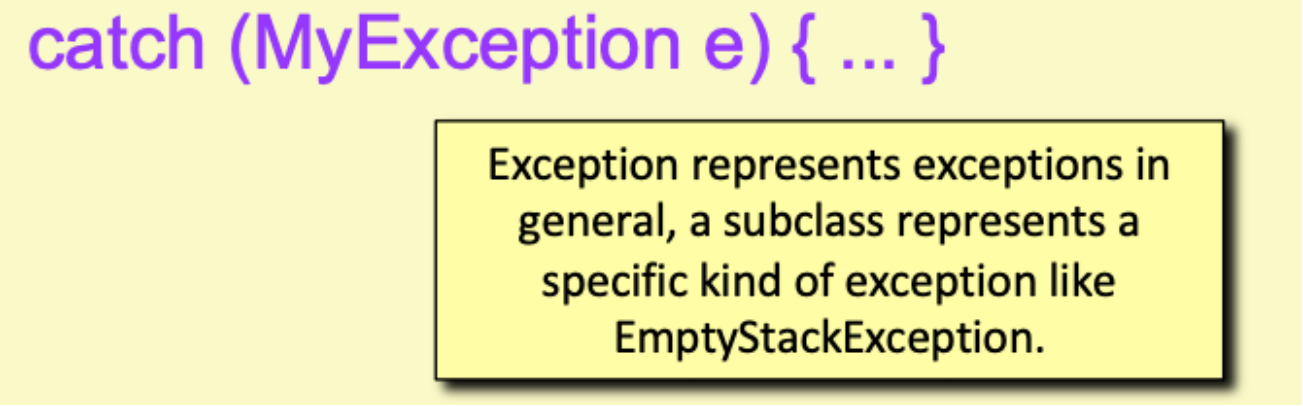
try and catch



throw

- throw new MyException (“Method failed”);
- The throw statement throws an exception object.
- It takes an exception object reference as an argument.
- In the chain of active method calls there must be a method with a catch block to catch the exception.

catch

- catch (Exception e) { ... }
- This catches an object of library class Exception or any of its subclasses.
- Typically, your exceptions are subclasses of class Exception.
- 

finally

```
try
{
    if (f()) g(); // code with method calls
    h();
}
catch (MyException e) // optional
{
  // Do something if exception thrown
}
finally
{
  // Guaranteed to execute
  // this whatever happens.
}
```

A finally block will be *always* be evaluated regardless of what else happens, before the enclosing method terminates.

Try-finally, with no catch is also allowed. This will guarantee that the code in the finally block will run whatever happens in the try block (assuming the program doesn't crash).

exception classes

- Throwable
  - Superclass of all exception classes.
- Error (extends Throwable)
  - Serious error that is not usually recoverable.
- Exception (extends Throwable)
  - Error that *must* be caught and recovered from.
- RuntimeException (extends Exception)
  - Error that *may* be caught, not try/catch required.

writing exception classes

```
class MyException extends Exception
{
  public MyException ()
  { super("Default message") ; }

  public MyException (String s)
  { super (s) ; }
}
```

The extends keyword specifies that MyException is a *subclass* of Exception.

We will be looking at subclasses in detail later in the module.

example:

```
public void f(String s)   // s should represent an integer.
{
    int n ;
    try
    {
       n = Integer.parseInt(s) ; // This can fail
    }
   catch (NumberFormatException e)
   {
        n = -1 ; // Set n to some default value - is this a good
                 // idea?
   }
   // Carry on and use n
}
```

If String s does not contain characters representing an int, there is *no* valid result parseInt can return. There is no int value that is not an int!

difference between throw, throws, catch, finally, try:

In Java, error handling is managed using exceptions, which are handled by several keywords such as `try`, `catch`, `finally`, `throw`, and `throws`. These keywords form the basis of

Java's exception handling framework, allowing developers to manage errors gracefully. Here's a breakdown of each keyword and its usage:

1. `try`

The `try` block is used to encapsulate a block of code that might throw an exception. The code inside the `try` block is monitored by Java's runtime system for any exceptions. If an exception occurs, the runtime system looks for the nearest handler, which is the `catch` block associated with it.

`try {// code that might throw an exception }`

2. `catch`

The `catch` block is used to handle the exception thrown within the `try` block. Each `catch` block is an exception handler that handles the type of exception indicated by its argument. The argument type should be a subclass of `Throwable`. The `catch` block only executes if an exception of the specified type occurs within the associated `try` block.

`catch (ExceptionType name) { // code to handle the exception }`

3. `finally`

The `finally` block follows `try` or `catch` blocks and will execute whether or not an exception is thrown. If an exception is thrown, the `finally` block will execute even if no catch block handles the exception. It is typically used for clean-up activities like closing file streams or releasing resources.

```
finally {
    // code to be executed after try block ends
}
```

4. `throw`

The `throw` keyword is used within a method to throw an exception. A method uses `throw` to indicate that it cannot handle a situation, and it wishes to pass the responsibility of handling a particular error/exception to the caller of the method. The exception object to be thrown must be an instance of `Throwable` or a subclass thereof.

```
throw new ExceptionType("Error Message");
```

5. `throws`

The `throws` keyword is used in the method signature to indicate that this method might throw one or more exceptions. It does not throw the exceptions but merely advertises to the method's callers that this method might throw these exceptions. Callers of this method must handle these exceptions.

```
public void methodName() throws IOException, SQLException {
// code that might throw IOException or SQLException
}
```

Unit Testing

concept:

- Create an object of a class being tested.
- Call a method.
- Check the value returned.
- Or if a void method, call another non-void method and check the value returned.
    - Observing the state of object.

run tests with JUnit