0. CONTEXT 3

1. COLLECTIONS 3

1.1. SUMMARY 3

2. TEST DRIVEN DEVELOPMENT 4

2.1. OVERVIEW 4
2.2. JUNIT 4
2.2.1. ASSERTIONS 4
2.2.2. JUNIT ANNOTATIONS 5
2.2.3. JUNIT EXAMPLE 5
2.3. MOCKITO 9
2.3.1. OVERVIEW 9
2.3.2. VERIFICATION (HOW MOCKITO WORKS) 9
2.3.3. MOCKITO EXAMPLE 10

3. EXCEPTIONS (TRY… CATCH) 13

3.1. OVERVIEW 13
3.2. TRY AND CATCH 13
3.3. FINALLY 14
3.4. THROW 14
3.5. MULTIPLE CATCH BLOCKS AND THEIR ORDER: 15
3.6. STACK TRACE: 15
3.7. HIERARCHY: 15
3.8. THROW VS THROWS: 15

4. CLASSES & ABSTRACT CLASSES 17

4.1. CONCRETE CLASSES 17
4.2. CONSTRUCTORS 18
4.3. ABSTRACT CLASSES & METHODS 19
4.4. INTERFACES 20
4.5. OVERRIDING METHODS 22
4.6. OVERLOADING 23
4.7. SUPER AND THIS 25

5. THE 4 PILLARS OF OOD (ARSALAAN) ... 25

5.1. PILLAR 1: 25
5.2. PILLAR 2: 25
5.3. PILLAR 3: 26
5.4. PILLAR 4: 26

6. JDBC (JAVA DATABASE CONNECTIVITY, JAVA+SQL) [AMANDA] 27

6.1. BACKGROUND & SETUP: 27
6.2. JDBC INTERFACES 27
6.2.1. CONNECTION: 27
6.2.2. STATEMENT: 27
6.2.3. RESULTSET: 28
6.2.4. PREPAREDSTATEMENTS (EXTENDS STATEMENT): 28
6.2.5. JDBC EXAMPLE CODE 29
6.2.6. CALLABLESTATEMENT (EXTENDS PREPAREDSTATEMENT): 30
6.2.7. STOREDPROCEDURE: 30

7. JPA (JAVA+DATABASES) (JACK) 31

7.1.1. BACKGROUND 31
7.2. METHODS: 31
7.2.1. FIND: 31
7.2.2. PERSIST: 31
7.2.3. MERGE: 31
7.2.4. REMOVE: 31
7.2.5. TYPED QUERY: 32
7.2.6. ANNOTATIONS: 32
7.2.7. ENTITYMANAGER & BEGIN() COMMIT() .. 32

8. SERVLET (CHARLIE) 33

8.1.1. BACKGROUND & OLD METHODS: 33
8.1.2. WEBSITE STATES: KEEPING INFORMATION ABOUT THE USER 33
8.1.3. LIFECYCLE OF A REQUEST MADE TO A SERVER: 34
8.1.4. LIFECYCLE OF A SERVLET: (WOULDN'T NORMALLY SEE) 34

9. JSP (JAVA + HTML) (DAYA) 35

9.1. JSP OVERVIEW 35
9.2. JSP DIRECTIVE 35
9.3. EXPRESSION LANGUAGE 36
9.4. DECLARATION 37
9.5. SCRIPTLET 37
9.6. EXPRESSION 37
9.7. LIFECYCLE: 38
9.8. JSTL TAGS 38
9.9. JSTL BENEFITS 39

10. MEMORY (MO) 40

10.1. EXAMPLE OF PASS-BY-VALUE LANGUAGE . 40
10.2. STACK: 40
10.3. HEAP: 40
10.4. EXAMPLES OF STACK AND HEAP: 40
10.5. NUMBER OF STACKS AND HEAPS IN JAVA . 40

10.6. STACK OVERFLOW ERROR & EXAMPLE 41
10.7. GARBAGE COLLECTOR 41

11. DESIGN PATTERNS (WILL) 42

11.1. 1. OBJECT POOL 42
11.2. 2. FACTORY 42
11.3. 3. SINGLETON 42
11.4. 4. ADAPTER 42
11.5. 5. COMMAND 42
11.6. 6. OBSERVER 42

12. THREADS (ASH) 44

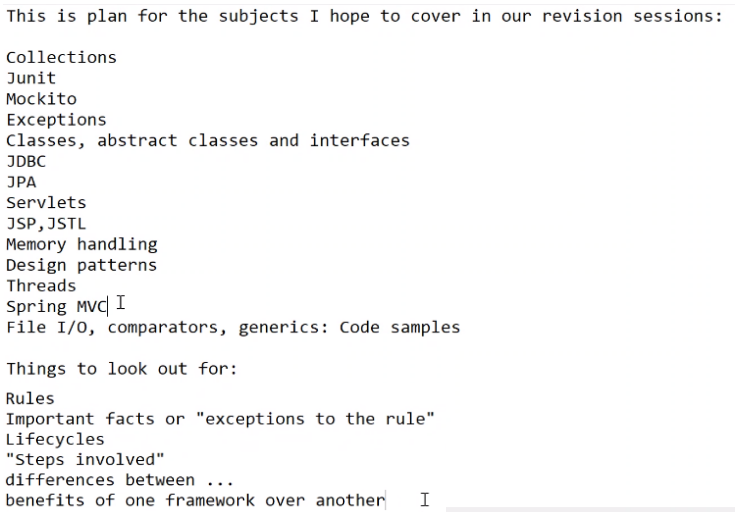
12.1. 11.1 BENEFITS OF MULTI-THREADING 44
12.2. 11.2 MAKING THREADS 44
12.3. 11.3 THREAD METHODS 44
12.4. 11.4 MULTI-THREADING HAZARDS AND SOLUTIONS 45
12.4.1. 11.4.1 HAZARDS CAUSED BY LOCKS AND SYNCHRONIZATION 45

13. COMPARISONS 46

13.1. COMPARABLE 49
13.2. COMPARATOR 49

0. Context

10 Questions.
30 Minutes.
8Q = Theory
2Q = Reading Out Pseudo Code
“Closed Book”
Will cover OOD and Java course.
Will NOT ask about Java 8 subjects.



1. Collections

The *benefit of using a Collection is that they have methods*. Java provides a Collections Framework with some classes and interfaces, these include:

Collections Framework

Interfaces	Classes
List	ArrayList, Vector
Set	HashSet, TreeSet
Map	HashMap, TreeMap

1.1. Summary

Type	Description	Allows Duplicates?	Ordered?	Example
List	Like a shopping list.	Y	Y	ArrayList, Vector
Set	Opposite to List.	N	N	TreeSet, HashSet
Queue	First In, First Out (FIFO). Like a bus line.	Y	Y	PriorityQueue
Stack	Opposite of Queue. Last in, First out (LIFO). Like stacking and taking plates.	Y	Y	
Map	Stores items in "key/value" pairs, and you can access them by an index of another type.	Keys = N Values = Y	HashMap = N TreeMap = Y	HashMap, TreeMap

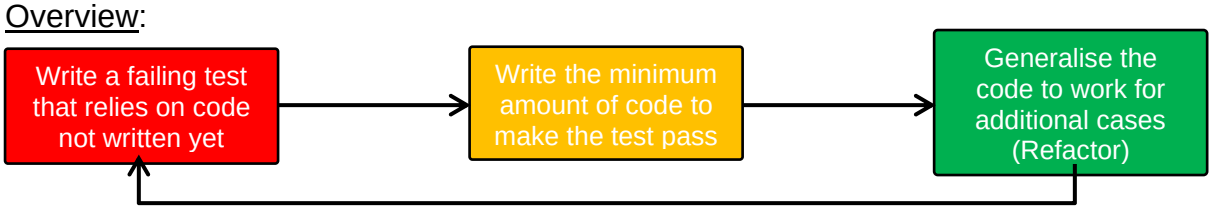
Map: Keys and Value’s can be of different types OR the same type. Must specify their types when creating Map objects.

2. Test Driven Development

2.1. Overview

"Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring). It is a technique where the **programmer writes a test before any production code, and then writes the code that will make that test pass.**

Overview:



The 5 Steps of TDD

1. Write the test
2. Make the test compile
3. Watch the test fail
4. Do just enough to get the test to pass
5. Refactor and generalise

Qualities of a Good Test

Focused - It should only test one thing.
Easy to read - The test name should make it clear what the test is doing.
Simple - Don't overcomplicate your test with loops and decisions.
Independent - Individual tests should not affect each other in any way.
Flexible - A test and the code it is testing should be able to be re-used in different projects without having to change anything.

2.2. JUnit

The pattern for writing unit tests is:

- Arrange – Set up preconditions
- Act – Call the code under test
- Assert – Check that expected results have occurred

2.2.1. Assertions

They are methods and **can verify whether a test should pass or fail by checking its return value**. Below are examples of some common assertions:

```
assertEquals(expectedObject, actualObject)

assertNotEquals(expectedObject, actualObject)

assertTrue(boolean)

assertNull(object)

assertFalse(boolean)
```

assertNotNull(object)

assertArraryEquals(expectedArray,actualArray)

2.2.2. JUnit Annotations

It is a special form of syntactic meta-data that can be added to Java source code for *better code readability and structure*. Variables, parameters, packages, methods and classes can be annotated. Below are examples of some common annotations:

@Test - Indicates that a method is a test

@BeforeEach - The annotated method should be run before each test

@BeforeAll - The annotated method should be run once before all tests

@AfterEach - The annotated method should be run after each test

@AfterAll - The annotated method should be run once after all tests

2.2.3. JUnit Example

Step 1 – Enable JUnit5 (Section 18.1.1)
Step 2 – Create JUnit Test Case File (Section 18.1.2) [Select setUp() method stub]

GradeCalculatorServiceTest.java

```
package com.fdmgroup.tdd.gradecalculator;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class GradeCalculatorServiceTest {

	@BeforeEach
	void setUp() throws Exception {
	}

	@Test
	void test() {
		fail("Not yet implemented");
	}

}
```

Step 3 – Write an object of the 'ghost' class as an attribute. This will prompt you to create the actual class you have referenced.

GradeCalculatorServiceTest.java

```
class GradeCalculatorServiceTest{
	GradesClass gradesClass;
	@BeforeEach
```

```
	void setUp() throws Exception {
	}
	@Test
	void test() {
		fail("Not yet implemented");
	}
}
```

Step 4 – Ensure 'Source folder' of the Java Class being created is situated in the src/main/java directory.

GradesClass.java

```
public class GradesClass {

}
```

Step 5 – Create the object in the Test Case file in the setUp() method to finalise the setting up process (Arrange step).

GradeCalculatorServiceTest.java

```
class GradeCalculatorTest {

	GradesClass gradesClass;

	@BeforeEach
	void setUp() throws Exception {
		gradesClass = new GradesClass(); //Arrange
	}

	@Test
	void test() {
		fail("Not yet implemented");
	}

}
```

Step 6 – Now you must write the GradesClass method according to the Problem and carry out unit tests to **ensure the program not only passes the tests but passes a range of tests consistently and with minimal coding**.

GradesClass.java

```
public class GradesClass implements GradeCalculatorService {

	@Override
	public String getClassification(double mark) {

		if (mark < 75 && mark >= 0) {
			return "fail";
		}

		else if (mark >= 75 && mark < 80) {
			return "pass";
		}

		else if (mark >= 80 && mark < 90) {
			return "merit";
		}

		else if (mark >= 90 && mark <= 100) {
```

```
            return "distinction";
        }

        return null;
    }
}
```

GradeCalculatorServiceTest.java

```
class GradeCalculatorServiceTest {

	GradesClass gradesClass;

	@BeforeEach
	void setUp() throws Exception {
		gradesClass = new GradesClass(); //Arrange
	}

	// Testing Fail
	@Test
	void testGradeCalculatorService_returnsFail_whenPassed55() {
		String mark = gradesClass.getClassification(55); // Act
		assertEquals("fail", mark);
	}

	@Test
	void testGradeCalculatorService_returnsFail_whenPassed0() {
		String mark = gradesClass.getClassification(0); // Act
		assertEquals("fail", mark);
	}

	// Testing Pass
	@Test
	void testGradeCalculatorService_returnsPass_whenPassed75() {
		String mark = gradesClass.getClassification(75); // Act
		assertEquals("pass", mark);
	}

	@Test
	void testGradeCalculatorService_returnsPass_whenPassed79() {
		String mark = gradesClass.getClassification(79); // Act
		assertEquals("pass", mark);
	}

	// Testing Merit
	@Test
	void testGradeCalculatorService_returnsMerit_whenPassed80() {
		String mark = gradesClass.getClassification(80); // Act
		assertEquals("merit", mark);
	}

	@Test
	void testGradeCalculatorService_returnsMerit_whenPassed89() {
		String mark = gradesClass.getClassification(89); // Act
		assertEquals("merit", mark);
	}

	// Testing Distinction
	@Test
	void testGradeCalculatorService_returnsDistinction_whenPassed90() {
		String mark = gradesClass.getClassification(90); // Act
		assertEquals("distinction", mark);
	}
	@Test
	void testGradeCalculatorService_returnsDistinction_whenPassed100() {
		String mark = gradesClass.getClassification(100); // Act
		assertEquals("distinction", mark);
	}

	// Testing null
	@Test
```

```
    void testGradeCalculatorService_returnsNull_whenPassedminus20() {
        String mark = gradesClass.getClassification(-20); // Act
        assertNull(mark);
    }
    @Test
    void testGradeCalculatorService_returnsNull_whenPassed120() {
        String mark = gradesClass.getClassification(120); // Act
        assertNull(mark);
    }
}
```

2.3. Mockito

2.3.1. Overview

When testing in JUnit we test methods which return values. If we have *methods which have a return type of void* then we cannot use JUnit to test these methods. Instead, we can use Mockito.

Mocks let us test for interactions between components. Mockito uses mock objects. Mock objects **mimic the behaviour of real objects**. When we test using Mockito we test one class at a time. NEVER mock the class under test.

2.3.2. Verification (How Mockito Works)

Verification is the **foundation of testing in Mockito**. We can test methods which return void, checking they are being called the correct number of times. We can check that a method is called on a mock object, how many times it is called and what arguments are being passed to it.

Using Verification

Lets imagine we have the following program code:

```
mockObject.myMethod("one");
```

We can verify this is being called the correct number of times using the following code:

```
verify(mockObject, times(1)).myMethod("one");
```

Lets look at the separate parts:

- The verify method is in the Mockito framework
- The variable mockObject is the mock object you created
- times(1) allows you to check how many times myMethod has been called. Here we are checking it was called once.
- The method times(..) is in the Mockito framework.
- times(3) means the test expects the method to be called 3 times.

2.3.3. Mockito Example

Basket.java

```
import java.util.List;

public interface Basket {

	List<Book> getBooksInBasket();

	void addBook(Book book);

	void removebook(Book book);
}
```

Book.java

```
public class Book {

	private double price;

	public double getPrice() {
		return price;
	}

	public void setPrice(double price) {
		this.price = price;
	}
}
```

Checkout.java

```
import java.util.List;

public class Checkout {

	public double calculatePrice(Basket basket) {
		List<Book> books = basket.getBooksInBasket();

		double total = 0;

		for (Book book : books) {
			total += book.getPrice();
		}

		return total;
	}

	public void emptyBasket(Basket basket) {
		List<Book> books = basket.getBooksInBasket();

		for (Book book : books) {
			basket.removebook(book);
		}
	}

}
```

CheckoutTest.java

```
import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import static org.mockito.Mockito.*;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

@ExtendWith(MockitoExtension.class) // creates mock objects for everything marked @Mock
class TestCheckout {

	// The class we're testing is the only real object in the whole testcase
	Checkout checkout;

	// All other objects are mock objects:
	@Mock
	Basket mockBasket;

	@Mock
	Book mockBook1, mockBook2, mockBook3;

	@BeforeEach
	void setUp() throws Exception {
		checkout = new Checkout();
	}

	@Test
	void test_calculatePrice_callsMockBaskets_getBooksInBasketMethod() {
	    checkout.calculatePrice(mockBasket); // We don't care about the return value!
	    verify(mockBasket).getBooksInBasket(); // passes if getBooksInBasket was called
	}

	@Test
	void test_calculatePrice_returns20point75WhenBooksCosting9point5and11point25Passed() {
		when(mockBook1.getPrice()).thenReturn(9.5);
		when(mockBook2.getPrice()).thenReturn(11.25);
		List<Book> mockBooks = new ArrayList<>(Arrays.asList(mockBook1,mockBook2));
		when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);

		double total = checkout.calculatePrice(mockBasket); // Act
		assertEquals(20.75,total); // Assert
	}

	@Test
	void test_calculatePrice_returns25WhenBooksCosting9pnt5_11pnt25_pnt25PassedIn() {
		when(mockBook1.getPrice()).thenReturn(9.5);
		when(mockBook2.getPrice()).thenReturn(11.25);
		when(mockBook3.getPrice()).thenReturn(4.25);
		List<Book> mockBooks = new
		ArrayList<>(Arrays.asList(mockBook1,mockBook2,mockBook3));
		when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);

		double total = checkout.calculatePrice(mockBasket); // Act
		assertEquals(25.0,total); // Assert
	}

	@Test
	void test_emptyBasket_callsRemoveBook2Times_whenBasketWith2BooksPassedIn() {
		List<Book> mockBooks = new ArrayList<>(Arrays.asList(mockBook1,mockBook2));
		when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);
```

```
        checkout.emptyBasket(mockBasket);
        verify(mockBasket, times(2)).removeBook(any(Book.class)); // passes if remove
        book is called twice
    }

    @Test
    void test_emptyBasket_callsRemoveBook3Times_whenBasketWith3BooksPassedIn() {
        List<Book> mockBooks = new
        ArrayList<>(Arrays.asList(mockBook1, mockBook2, mockBook3));
        when(mockBasket.getBooksInBasket()).thenReturn(mockBooks);
        checkout.emptyBasket(mockBasket);
        verify(mockBasket, times(3)).removeBook(any(Book.class)); // passes if remove
        book is called three times
    }

}
```

3. Exceptions (Try… Catch)

3.1. Overview

When executing Java code, different errors can occur such as coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: **Java will throw an exception** (throw an error).

3.2. Try and Catch

The `try` statement allows you to define a block of **code to be tested** for errors while it is being executed. The `catch` statement allows you to define a block of **code to be executed if an error occurs** in the try block. The `try` and `catch` keywords come in pairs:

Try and Catch Syntax:

```
try {
  // Block of code to try
}
catch(Exception e) {
  // Block of code to handle errors
}
```

Example of Error Catching:

```
public class Runner {

    public static void main(String[] args) {

        try {
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]);
        }
        catch (Exception e) {
            System.out.println("Something went wrong.");
        }

    }
}
```

This will generate an error, because myNumbers[10] does not exist.

Now the output will be: 'Something went wrong.'

As opposed to the generic error message.

3.3. Finally

The `finally` statement lets you **execute code after** `try...catch`, regardless of the result:

Example of using Finally:

```
public class Runner {

	public static void main(String[] args) {

		try {
			int[] myNumbers = {1, 2, 3};
			System.out.println(myNumbers[10]);
		} catch (Exception e) {
			System.out.println("Something went wrong.");
		} finally {
			System.out.println("The 'try catch' is finished.");
		}

	}
}
```

Now the output will be:
'Something went wrong.'
'The 'try catch' is finished.'

3.4. Throw

The `throw` statement allows you to **create a custom error** (red error message). The throw statement is used together with an exception type.

There are many exception types available in Java: `ArithmeticException`, `FileNotFoundException`, `SecurityException`, etc:

Example of using Throw:

```
public class Runner {

	static void checkAge(int age) {

		if (age < 18) {
			throw new ArithmeticException("Access denied - You must be at least 18
			years old.");
		}
		else {
		    System.out.println("Access granted - You are old
			enough!");
		}

	}

	public static void main(String[] args) {

		checkAge(15); // Set age to 15 (which is below 18...)
```

Now the output will be:
Exception in thread "main"
java.lang.ArithmeticException:
Access denied – You must be at least 18…

Extra notes

3.5. Multiple catch blocks and their order:

```
try{
        //code which might cause an exception

} catch(FileNotFoundException fnfe){
        // catch the exception and deal with it
        fnfe.printStackTrace();

} catch(IOException ioe){
        // catch the exception and deal with it

} catch(SQLException sqle){
        // sql problem

} catch(Exception e){
        // BAD PRACTICE (too general)

} finally{
        // clean up resources (such as releasing connections)
```

FileNotFoundException inherits from IOException, therefore must go before IOException in the order.

Specific Exceptions @ Top
General Exceptions @ Bottom

If the two exception classes are not inherited from another then the order is not problematic.

Multiple catch blocks allow us to deal with different exceptions in different ways.

3.6. Stack Trace:

```
try{
  openFile("C:\\ade.txt");
} catch(IOException ioe){
  ioe.printStackTrace();
}
```

printStackTrace(); → useful tool for diagnosing an exceptions. It tells you what happened and where in the code this happened.

3.7. Hierarchy:

Throwable	
Error	Exception
RuntimeException (bad coding practices like divide by 0)	Non Runtime Exceptions (caused by circumstances)

3.8. Throw vs Throws:

```
public vo        id openFile(String filename) throws IOException{ // MIGHT raise exception

  if(....){
```

```
    throw new IOException(); // generates the exception
  }
}
```

Throws=used to declare an exception, which means it works similar to the try-catch block.
Throw=used to throw an exception explicitly.

4. Classes & Abstract Classes

4.1. Concrete Classes

Java is an object-oriented programming language. Everything in Java is associated with classes and objects, along with its attributes (variables) and methods. For example: a car is an object. The car has attributes, such as weight and colour, and methods, such as drive and brake. A Class is like an object constructor, or a "blueprint" for creating objects. To create a class, use the keyword `class`:

Creating a Class:

Notes from video:
Concrete classes/Non abstract classes:
Can have:
-member variables
-non abstract methods
-constructors
Abstract classes:
Can have:
-member variables
-non abstract methods
-constructors
-abstract methods: once one abstract method appears in a class then the whole class has to be abstract. No code in abstract methods.
-Cannot create instances of abstract classes.
-Constructor can be called from constructor in a sub class. And it would be called by using the super keyword.

```
package com.fdmgroup.newPackage;

public class Car {

	// Attributes
	String companyName;
	String modelName;
	int year;
	int weight;
	String colour;

	// Behaviours/Methods (Functions)
	public void drive() {
		System.out.println("Car is driving.");
	}

	public void brake() {
		System.out.println("Car is braking.");
	}
}
```

In Java, an object is created from a class. We have already created the class named Car, so now we can use this to create objects. To create an object of Car, specify the class name, followed by the object name, and use the keyword `new`.

Creating an Object:

```
package com.fdmgroup.newPackage;
public class Runner {
	public static void main (String[] args) {

	Car car1 = new Car(); // Empty Object 1
	Car car2 = new Car(); // Empty Object 2
	Car car3 = new Car(); // Empty Object 3

	// Adding Attributes to Object 1
	car1.companyName = "Ford";
	car1.modelName = "Fiesta";
	car1.year = 2013;
```

Think of this as the data type.

Object name

Creating the empty object by calling the class

```
car1.weight = 1200;
car1.colour = "White";

// Calling the drive() method from the Car class for Object 1
car1.drive();
```

4.2. Constructors

The format for creating an object for Car is similar to the one for creating ArrayList.

All objects follow this format.

Creating a new object using the approach in the example abov
you have multiple objects. We can create a custom construct
initialise an objects attributes.

Creating a Constructor:

```
package com.fdmgroup.newPackage;

public class Car {

	// Attributes (Variables & Constants)
	String companyName;
	String modelName;
	int year;
	int price;
	int weight;
	String colour;

	// Constructor
	public Car(String companyName, String modelName, int year, int p		rice, int weight,
String colour) {
		super();
		this.companyName = companyName;
		this.modelName = modelName;
		this.year = year;
		this.price = price;
		this.weight = weight;
		this.colour = colour;
	}

	// Behaviours/Methods (Functions)
	public void drive() {
		System.out.println("Car is driving.");
	}

	public void brake() {
		System.out.println "Car is braking.");
	}

}
```

Creating an Object (after constructor added):

```
package com.fdmgroup.newPackage;
```

```
public class Runner {

	public static void main (String[] args) {

	Car car1 = new Car("Ford", "Fiesta", 2013, 5000, 1200, "White");

	System.out.println(car1.modelName);

	}

}
```

Data abstraction is **the process of hiding certain details and showing only essential information to the user**. Abstraction can be achieved with either abstract classes or interfaces.

4.3. Abstract Classes & Methods

The abstract keyword is a non-access modifier, used for classes and methods:

- Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Why and when to use Abstract Classes and Methods? To achieve security - hide certain details and only show the important details of an object.

Creating an abstract class (with both abstract and regular methods):

Animal.java:

```
public abstract class Animal {

	// Abstract method (no body)
	public abstract void animalSound();

	// Regular method
	public void sleep() {
		System.out.println("ZzzZzzZZzz");
	}
}
```

Accessing an abstract class and overriding the abstract method:

Dog.java:

```
public class Dog extends Animal {

	@Override
```

```
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The dog says: woof woof");
    }
}
```

Runner.java:

```
public class Runner {

    public static void main(String[] args) {

    // Creating objects
    // Cannot create "Animal myAnimal = new Animal();" object from abstract class
    Animal myDog = new Dog();

    // Calling the overridden abstract method and regular method from child class.
    myDog.animalSound();
    myDog.sleep();

    }
}
```

4.4. Interfaces

Another way to achieve abstraction in Java, is with interfaces. An interface is a **completely "abstract class" that is used to group related methods with empty bodies**:

Creating an Interface Example:

```
interface Animal {

    // interface methods (no body)
    public void animalSound();
    public void run();

}
```

To access the interface methods, the interface must be "implemented" (similar to inherited) by another class with the `implements` keyword (instead of extends). **The body of the interface method is provided by the "implement" class**:

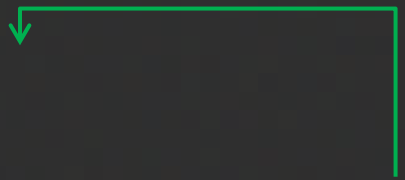
Implementing an Interface Example:

```
// Interface
interface Animal {

        // interface method (does not have a body)
        public void animalSound();
        public void sleep();

}

// Bird "implements" the Animal interface
class Bird implements Animal {

        public void animalSound() {
                // The body of animalSound() is provided here
                System.out.println("The bird says: tweet tweet");
        }

        public void sleep() {
                // The body of sleep() is provided here
                System.out.println("Zzz");
        }
}

// Main method
class Main {

        public static void main(String[] args) {
                // Create a Bird object
                Bird myBird = new Bird();
                myBird.animalSound();
                myBird.sleep();
        }
}
```

Why and when to use Interfaces?
1) To achieve security - hide certain details and only show the important details of an object (interface).
2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces because the class can implement multiple interfaces.

Multiple Interfaces:

```
interface FirstInterface {
        public void myMethod(); // interface method
}

interface SecondInterface {
        public void myOtherMethod(); // interface method
```



```
}

class DemoClass implements FirstInterface, SecondInterface {
    public void myMethod() {
        System.out.println("Some text..");
    }
    public void myOtherMethod() {
        System.out.println("Some other text...");
    }
}

class Main {
    public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
    }
}
```

To implement multiple interfaces, separate them with a comma

NOTE:
Interfaces cannot be used to create objects. Interface methods do not have a body. On implementation of an interface, you must override all of its methods. Interface methods are by default `abstract` and `public`. Interface attributes are by default `public`, `static` and `final`. An interface cannot contain a constructor.

4.5. Overriding Methods

This is when **code in the child class method completely replaces the code from the parent class method** that it overrides.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Cats, Dogs, Birds etc. – and they also have their own implementation of an animal sound (cat meows, dog barks, etc.):

Animal.java

```
public class Animal {

    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}
```

Dog.java

```
public class Dog extends Animal {

    public void animalSound() {
        System.out.println("The dog says: woof woof");
    }
}
```

Use extends keyword to inherit from a class

Cat.java

```
public class Cat extends Animal {

        @Override
        public void animalSound() {
                System.out.println("The cat says: meow");
        }
}
```

Can add @Override annotation to check method is being overridden.

Runner.java

```
public class Runner {

        public static void main(String[] args) {

        // Creating objects
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        // Calling the same method() for each class.
        myAnimal.animalSound();
        myDog.animalSound();
        myCat.animalSound();
```

Each print out different statements!

NOTE:

- To access a method from the parent class (that has been overridden in the child class), we must use the `super` keyword (as shown in 7.1.1 Super keyword section).
- Using the `final` keyword in a method header will prevent the method from being overridden in any child classes (if `final` already stated in class header, then no need for `final` in method header).
- Overriding the `equals()` and `hashcode()` methods is sometimes necessary to ensure they behave as 'expected' when comparing objects. Eclipse has a built-in option to enable this feature.

4.6. Overloading

This is when a class has **multiple methods with the same name but with different arguments**. Each version of the method will have slightly different functionality. The version called depends on the arguments passed in.

Overloading Method Example:

```
public class Car {
        public void accelerate() {}
```

accelerate(40) will call the first method accelerate(2.5) will call the second method

```
    public void accelerate int speedLimit {}

    public void accelerate double hillGradient {}
}
```

Overloading Constructors:

```
public class Car {

    public Car() {}

    public Car(String model) {}
}
```

A car object can now be created by calling either constructor:
Car car1 = new Car();
Car car2 = new Car("Tesla");

Horizontal Constructor Chaining:

```
public class Car {

    public Car(int maxSpeed){
        // code to check maxSpeed is valid number
    }

    public Car(int maxSpeed, String model) {
        this(maxSpeed);
        this.model = model;
    }
}
```

Arguments passed to a different constructor in the same class using this().

this() used to avoid code duplication (instead of copying code from first constructor again)

4.7. Super and this

this. calls a variable or method in this class from anywhere in this class
this() calls a constructor in this class but you must call it inside a constructor

```
class BankAccount{

int accountNumber;
String accountName;
double balance;

  BankAccount(int accountNumber, String accountName, double balance){
    // 3 in the full way
  }

  BankAccount(int accountNumber, String accountName){
    this(accountNumber, accountName, 0.0);

  in a constructor, the first statement must be EITHER super(.....); or this(......);
    super(); // implicit call to super()
  }

}
```

super. calls a variable or method in the super class
super() calls a constructor of the super class

5. The 4 Pillars of OOD (Arsalaan)

5.1. Pillar 1:

Abstraction: Extracting the relevant information for the required task.

5.2. Pillar 2:

Polymorphism: "Many forms"

e.g many ways to create a 'vehicle' object:

```
Vehicle vehicle = new Car();
..
vehicle = new Lorry();
```

Overloading (compile time) v Overriding (run time) polymorphism
Overloading = class has multiple methods with the same name but with different arguments. Each version of the method will have slightly different functionality.
e.g.

```
Car(){}
```

```
Car(int engineSize, int mpg){}
```

NOTE: The code `Car car = new Car();` would know at compile time which version was being called

Overriding = code in the child class method completely replaces the code from the parent class method that it overrides:

e.g.
Vehicle class: `public void drive()`
Car class: `public void drive()`

NOTE: The code `car.drive();` would know at runtime which version was being called

5.3. Pillar 3:

Inheritance: inheriting a class or implementing an interface.

5.4. Pillar 4:

Encapsulation: grouping logically related things, member variables together.

6. JDBC (Java database connectivity, Java+SQL) [Amanda]

6.1. Background & Setup:

JDBC is an API (application programming interface) ("a library of code") for Java that defines how a client can access a database / enables Java applications to interact with databases.

- Need to set up connections and code ourselves.
- Need to add dependency to pom.xml to download (h2) jar file into project.

Persistence = saving data permanently

- Persistence vs collections (ArrayLists, etc.): data in collections is removed when program exits so all objects are lost from memory, persist stores data to database.
- Advantage of database: portable (can send to other computers), there is vast storage space, quicker to query.

DriverManager = the only concrete class, use to install/set up a vendor specific driver (e.g. h2, oracle) and get a Connection object.

- New `org.h2.Driver` ⇒ "use h2 as database type" (pointing to class that looks after database).
- Need to set up JDBC: database vendor (h2), database location URL (~/connectionName) and Username+Password (sa, blank password).

```
//what to use to connect to database
DriverManager.registerDriver(new org.h2.Driver);
//get connection to database
Connection connection=DriverManager.getConnection("jdbc:h2:~
/connectionName","sa","");
```

6.2. JDBC Interfaces

6.2.1. Connection:

used to set up a connection between Java application and database. Vendor specific (we specified h2).

- Set using DriverManager:
  ```
  Connection connection =
  DriverManager.getConnection(connectionURL);
  ```
- remember to do `connection.close()` when finished!

6.2.2. Statement:

used to send SQL statements to database via queries (interact via connection)

- Created from connection object:
  ```
  Statement statement = connection.createStatement();
  ```

`statement.execute(s)`: CREATE/DROP/ALTER (change table structure), returns boolean (worked or not)

```
e.g. s = "DROP table tester IF EXISTS" or "CREATE table tester(mynum
number(6), myname varchar2(20))"
```

`statement.executeUpdate(s)`: INSERT/DELETE/UPDATE (change data in table), returns int (number of rows affect by query)
`e.g. s = "INSERT INTO test(mynum, myname) VALUES (1, 'Amanda')"`
`- NOTE SINGLE QUOTE MARKS`

`statement.executeUpdate("commit");` to enter data and save changes
`statement.executeQuery(s):` SELECT query, returns ResultSet object
`e.g. s = "SELECT mynum, myname FROM tester"`

6.2.3. ResultSet:

a table of data/results from database. Use SELECT statement

```
ResultSet resultSet = statement.executeQuery("SELECT myname,
mynum FROM tester");

while(resultSet.next(){
int mynumber=resultSet.getInt("mynumber") ;
String myname = resultSet.getString("myname");
System.out.print("" + mynumber+"," + myname);
}
```

- `resultSet.next()`: iterate through resultSet row by row (initially starting at nothing, then row 1), boolean → true if row has data.
- `resultSet.getInt(), .getString()`: getter methods to get data from row. Take column name or column index (start at 1).

NOTICE: HE SAID IN THE REVISON VIDEO HE MIGHT GET US TO WRITE 5-9LINES OF CODE SOMETHING LIKE SETTING UP RESULTSET, SHORT CODE SAMPLE

6.2.4. PreparedStatements (extends Statement):

write SQL code with ? placeholder for missing parameter values

- Safer alternative to statement objects.
  - Problem with Statement objects: the structure means if you want to put in some dynamic values, you need to switch between SQL and Java variables, so can be prone to error and SQL injection (hacking - someone can forcefully insert their own code).

```
//switching between SQL and Java variables (brokerId, firstName)
statement.executeQuery("Select brokerId, firstName from brokers
where brokerId ="+brokerId+" and firstName=" + firstName);
//more complex when needed to use single quotes for varchar2 types
(here brokerId is a String)
statement.executeQuery("Select * from brokers where brokerId='
"+brokerId+" ' ");
```

//prepared statement example

```
PreparedStatement ps = connection.prepareStatement("DELETE FROM
brokers WHERE id = ? and name = ?");
ps.setInt(1, id); //set for first ?
```

```
ps.setString(2, name); //set for second ?
ps.executeUpdate();
```

//another example

```
PreparedStatement preparedStatement = connection.
prepareStatement("Select ... from brokers where brokerId = ?
and firstName = ?");
preparedStatement.setInt(1, brokerId);
```

//.setDataType (int), position number of ? (1), value (brokerId)

```
preparedStatement.setString(2, firstName);
ResultSet resultSet = preparedStatement.executeQuery();
```

//using excuteQuery because it's a SELECT statement

// .. can do a while loop here to print resultSet in console

6.2.5. JDBC Example Code

//possible sqlException, may be a problem with h2 database, surround with try-catch..

```
try {

DriverManager.registerDriver(new org.h2.Driver());
Connection connection = DriverManager.getConnection(
"jdbc:h2:~/connectionName", "sa", "");

Statement statement = connection.createStatement();

statement.execute("DROP TABLE BROKERS IF EXISTS;");
statement.execute("CREATE TABLE BROKERS(brokerId NUMBER(6),
firstName VARCHAR2(20), lastName VARCHAR2(20));");
statement.executeUpdate("INSERT INTO BROKERS(brokerId,
firstName, lastName) VALUES(1,'Hi', 'Everyone');");

statement.executeUpdate("COMMIT");

ResultSet resultSet = statement.executeQuery("SELECT * FROM
BROKERS");

while (resultSet.next()) {
int brokerId = resultSet.getInt("brokerId");
String firstName = resultSet.getString("firstName");
String lastName = resultSet.getString("lastName");
System.out.println(brokerId+", "+firstName+", "+lastName);
}
```

//close connection. Best practice to close when something has been opened.

```
connection.close();

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

6.2.6. CallableStatement (extends PreparedStatement):

not supported by h2, used for Stored procedures, good for separating Java + SQL code.

6.2.7. StoredProcedure:

SQL code that you can save, so the code can be reused over and over again, just call it to execute it.

```
//stored procedure is "AddAUser", compile it first, then call it.
CallableStatement callableStatement =
connection.prepareCall("{call AddAUser(?,?,?)}"); //3 missing values
callableStatement.setInt(1, 1); //same as prepared statements
callableStatement.setString(2, "London");
callableStatement.setString(3, "Bridge");
callableStatement.executeUpdate(); //OR
ResultSet resultSet = callableStatement.executeQuery();
```

7. JPA (Java+Databases) (Jack)

7.1.1. Background

JPA Java Persistence API

- It is an Object Relational Mapping tool: ORM
- It maps Java Entities to SQL tables, so you don't have to write any SQL.
- Everything can be done in Java and hosted on the Java tier
- JPA improves upon JDBC Framework
- Centralizes all the connection details, login details into a persistence file, persistence.xml
- JPQL is a database independent language where you can write some code to query the database

It uses the EntityManager class to perform operations:
Remember to set EntityManagerFactory and EntityManager:

```
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("project");
EntityManager entityManager = entityManagerFactory.createEntityManager();
```

7.2. Methods:

7.2.1. find:

Looks up one object, using primary key:

```
public void addCustomer(Customer customer) {
Customer customerInDB = entityManager.find(Customer.class, customer.getUsername());
```

7.2.2. persist:

Saves a row to the database, after constructing object:

```
entityManager.persist(customer);
```

7.2.3. merge:

foreign key involved: BankAccount using FK from customer:

```
public void updateBankAccount(BankAccount newBankAccount) {
BankAccount accountInDB = entityManager.find(BankAccount.class, newBankAccount.getBankId());
EntityTransaction entityTransaction = entityManager.getTransaction();
entityTransaction.begin();
if (accountInDB == null) {
} else {
entityManager.merge(newBankAccount);
}
entityTransaction.commit();
```

7.2.4. remove:

Fully remove object from database:

```
public void removeBankAccount(int bankId) {
BankAccount accountInDB = entityManager.find(BankAccount.class, bankId);
EntityTransaction entityTransaction = entityManager.getTransaction();
entityTransaction.begin();
```

```
if (accountInDB == null) {
} else {
entityManager.remove(accountInDB);
}
entityTransaction.commit();
```

7.2.5. Typed Query:

TypedQuery<User> query = entityManager.createQuery(JPQL, User.class);
List<User> listUsers = query.getResultList(); provides list of users from database

Query query = entityManager.createNativeQuery(SQL) not recommended to use SQL
List<Object[]> List of object arrays - TypedQuery is a better method

This means you use Java methods to achieve database functions: select, update, insert

7.2.6. Annotations:

@Entity : Becomes table in database : @Entity(name="SC_CUSTOMERS") – Table Name
public class Customer {
@Id – Primary Key – Requires 1 per Entity
@ManyToOne – Easiest Foreign Key implementation

7.2.7. EntityManager & begin() commit()

EntityManagerFactory Used to create EntityManager
EntityTransaction begin() / commit() Change any data in the database :
begin() before you start to make any changes
commit() to save any changes

8. Servlet (Charlie)

8.1.1. Background & Old Methods:

Original methods and motivation for using Servlets

The most important methods for using a Servlet are doGet and doPost, they both have input of

```
doGet(HttpServletRequest request, HttpServletResponse response)
```

These allow it to obtain information from the request and to formulate a response. These methods are called from a .jsp page using the following tags:

```
<FORM action ="...." method="GET">
```

In most cases it is better to use POST as doGet sends the parameters through the address bar.

Majority of the time we will use request over response, request allows us to access information from the page and set attributes. Our most common use was:

```
request.getParameter("username");
request.getSession();
```

Redundant way of outputting to the screen

When we were very first working with webpages we used PrintWriter out = response.getWriter();

But this was very tedious as this is only useful for displaying information through Sys-out statements. We realised its much preferable to send the client to a new .jsp page.

8.1.2. Website States: Keeping information about the user

HTTP itself is stateless by default, which means it doesn't remember what has come before it.

This is where sessions come in handy. Session is a state management system stored on the server side. Cookies are a state management system stored on the client side.

For instance, we would want to store the username and basket in the session as this is personal to the user and allows us to customise pages accordingly.

```
HttpSession session = request.getSession();
session.setAttribute("username", username);
```

This is how we would control the session. To kill a session we simply use:

```
session.invalidate();
```

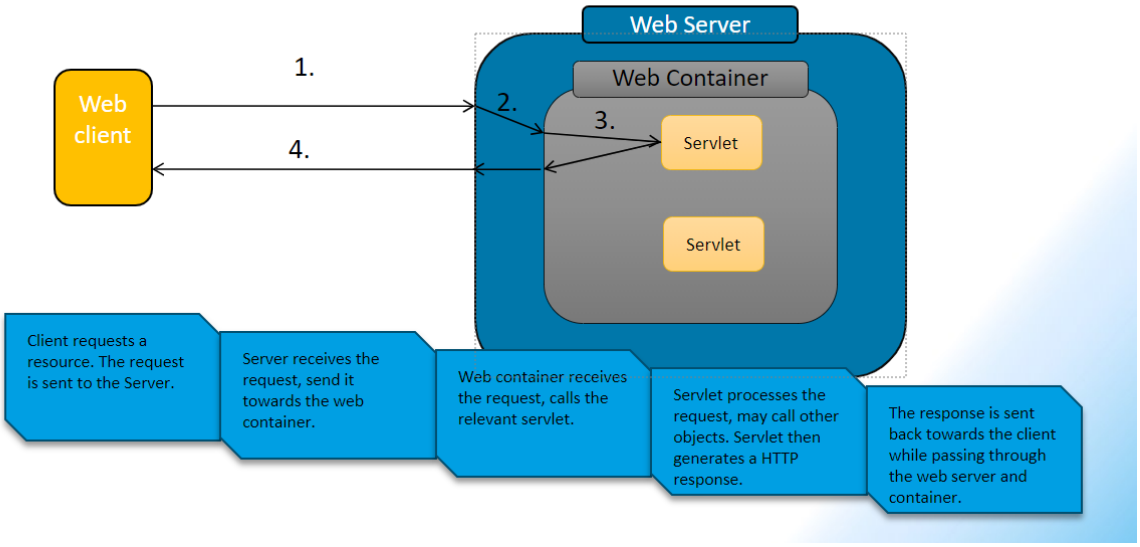
This kills the session which would be useful for a logout feature as the information about the user is forgotten.

RequestScope – Short term (between pages loading usually), **HttpSession** – medium length for the duration of the session and **ServletContext**- long term lasts until the server is restarted.

Informaton from ServletContext is global everyone can see it.

Typical use for it would be on a forum where you wish to see active users. Where it can then display a list of active users.

8.1.3. Lifecycle of a request made to a server:



8.1.4. Lifecycle of a servlet: (Wouldn't normally see)

init() - the servlet is initialised
service() - this is the most called function determines whether it is a doGet or doPost call.
destroy() - terminates the servlet (who woulda thunk it) :)

9. JSP (Java + HTML) (Daya)

9.1. JSP Overview

JavaServer Pages (JSP) is a technology for developing Webpages that supports dynamic content. This allows developers to embed java code in HTML pages by using specific JSP tags, the majority of which begin with <% and end with %>. A JavaServer Pages component is a type of Java servlet that acts as the user interface for a Java web application. JSPs can be written as text files that contain HTML as well as embedded JSP actions and commands.

JSP tags can be used for a variety of things, including retrieving data from a database or registering user preferences, passing control between pages, and sharing data between requests, pages, and so on.

9.2. JSP directive

The general structure of the servlet class is affected by a JSP directive. It's usually written like this:

Directive	
<%@page%>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
<%@include%>	Includes a file during the translation phase.
<%@taglib%>	Declares a tag library, containing custom actions, used in the page

The page directive is used to provide instructions to the container. These instructions pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page. Following is the basic syntax of the page directive:
<%@page attribute=”errorPage”%> // errorPage attribute defines the URL of another JSP that reports on Java unchecked runtime exceptions.

During the translation step, the include directive is used to include a file. During the translation process, this directive instructs the container to merge the content of other external files with the current JSP. Include directives can be placed anywhere in your JSP page.

The include directive's filename is actually a relative URL. The JSP compiler assumes that the file is in the same directory as your JSP if you just supply a filename without a path.

```
<%@include file="header.jsp" %>
```

The JavaServer Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behaviour.
The taglib directive states that your JSP page utilises a set of custom tags, specifies the library's location, and allows you to find the custom tags in your JSP page.

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%
>
<%@ taglib prefix="s"
uri="http://www.springframework.org/tags
"%>
<%@ taglib prefix="sf"
uri="http://www.springframework.org/tags
/form"%>
```

9.3. Expression language

JSP Expression Language makes it possible to easily access application data stored in JavaBeans components. JSP Expression Language allows you to create expressions both (a) arithmetic and (b) logical. Within a JSP Expression Language expression, you can use integers, floating point numbers, strings, the built-in constants true and false for boolean values, and null.

Null values are not displayed in the webpage.

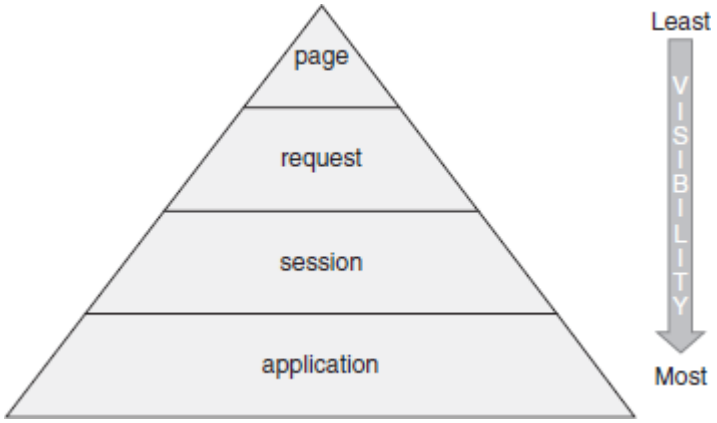
By using the code below in the controller

```
request.setAttribute("message", "Must
include a capital letter !");
```

You can store a variable called message and use it in a .jsp file.

```
${requestScope.message}
```

Implicit object	Description
pageScope	Scoped variables from page scope
requestScope	Scoped variables from request scope
sessionScope	Scoped variables from session scope
applicationScope	Scoped variables from application scope
pageContext	The JSP PageContext object for the current page



The pageScope, requestScope, sessionScope, and applicationScope variables provide access to variables stored at each scope level.

The pageContext object gives you access to the pageContext JSP object. Through the pageContext object, you can access the request object.

```
${pageContext.request.contextPath}
```

9.4. declaration

<%! %> not very useful

A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.

```
<%! int i = 0; %>
<%! int a, b, c; %>
 <%! Circle a = new Circle(2.0); %>
```

9.5. scriptlet

<% %> messy

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

```
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    Hello World!<br>
    <% out.println("Hello World”)); %>
</body>
</html>
```

9.6. expression

<%= %> use EL

A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.
Very similar to Expression Language but if the value is null, it will display null.

9.7. lifecycle:

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

when we change the JSP: all 7 steps take place:

1. translate JSP to servlet (All the JSP file content gets turned into out.print and out.write for various values.)
2. compile that servlet to a .class (same as normal java file)
3. load into memory (same as normal java file)
4. instantiate: create an instance of the class (same as normal java file)

If the JSP stays the same:

5. jspInit()

When a container loads a JSP it invokes the jspInit() method before servicing any requests. Initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

6. jspService()

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the _jspService() method in the JSP.
The jspService() method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e., GET, POST, DELETE, etc.

7. jspDestroy()

JSP is being removed from use by a container, the jspDestroy() method is the JSP equivalent of the destroy method for servlets.

9.8. JSTL tags

<c:if >

The <c:if > tag evaluates an expression and displays its body content only if the expression evaluates to true.

```
<c:set var="salary" scope="session" value="${2000*2}"/>
<c:if test="${salary > 2000}">
        <p>My salary is: <c:out value="${salary}"/><p>
</c:if>
```

<c:choose >, <c:when >, <c:otherwise >

The works like a Java switch statement in that it lets you choose between a number of alternatives.

```
<c:choose>
<c:when test="${salary <= 0}">
   …….
</c:when>
<c:when test="${salary > 1000}">
   ……..
</c:when>
<c:otherwise>
   ……
</c:other>
</c:choose>
```

<c:out >

The <c:out> tag is used to show the result of an expression. This is almost similar to the way <%= %> works. The distinction is that the <c:out> tag allows you to access properties using the simpler "." notation. For example, to access customer.address.street, use the tag <c:out value="customer.address.street"/>.

```
<c:out value="£ ${eachProduct.price}" />
```

<c:forEach >

<c:forEach> tags are similar to Java for, while, or do-while loop via a scriptlet. The tag is a commonly used tag because it iterates over a collection of objects.

```
<c:forEach var="eachProduct" items="${listOfProducts}">
    <TR><TD class="tabledata">
        <c:out value="${eachProduct.productName}" /> </TD>
  <TD class="tabledata"> <c:out value="£ ${eachProduct.price}" /> </TD>
  <td class="tabledata">
<sf:form
action="${pageContext.request.contextPath}/addProductToBasket/${eachProduct.prod
uctID}">
            <input type="submit" name="commit" value="add" />
       </sf:form>
 </td>
    </TR>
</c:forEach>
```

9.9. JSTL Benefits

1. Scriptlets are very messy and very hard to debug. Using JSTL avoids these messy scriptlets. Java and embedded HTML is not very readable
2. Web designers and web developers can make changes to a file containing JSTL.
Web designers will not recognize Java so cant fix scriptlets
Web developers might be too familiar with HTML

10. Memory (Mo)

10.1. Example of pass-by-value language

- Java is a pass by value language
- Litmus test is done to check if a language is a pass by value or pass by reference language.

```
void swap(int first, int second){
        int temp = first;
        first =second;
        second=temp;
}

int firstNumber=1;
int secondNumber=2;

swap(firstNumber,secondNumber);
                1               2
```

Swap method attempts to swap values around. However, it swaps the copies of the variables passed in, locally. So, the outcome is that no swap has happened, and the value of the variables would be the same.

Two types of memory

10.2. Stack:

when you call a method anything declared in there is stored as a stack, once the method is finished everything is cleared i.e., the stack is dismantled. If a new method is called a new stack is created for that. Stack stores local variables primitives (ints, floats, doubles etc) objective references and method references.

10.3. Heap:

Only stores objects.

10.4. Examples of Stack and Heap:

`String str = new String();` new String() : the String object -> HEAP
str : object reference -> STACK

`str=null;` you can do this to say that you're finished with this variable. A benefit is that the garbage collector knows that this variable Is no long being used

```
str = new String();
```

10.5. Number of stacks and heaps in Java

- There is only one heap per JVM(Java Virtual Machine)
- When you run a program in eclipse an environment is created (virtual machine) only one heap per environment.
- There can be many stacks. (One stack per thread)

10.6. **Stack overflow error & example**

Main method: there is a stack
method1: there will be another stack
method2: there will be another stack

```
int factorial(int value){
   return value * factorial(value-1); // create so many local variables on the stack
}
```

This method uses recursion to multiply the value it starts with the factorial of the number below. (value-1) Which means the method Is being called infinitely and there is no exit. Goes into the negative numbers. The stack created would overflow giving you a stack overflow error.

10.7. **Garbage collector**

- Is responsible for the heap.
- When memory usage reaches 85-95% the garbage collector must pause the program so it can clean out any objects that haven't been used in a while or when objects have been set to null.
- Garbage collection cannot be forced, it's automatic with Java.

`str=null;`	you can do this to say that you're finished with this variable. A benefit is that the garbage collector knows that this variable Is no long being used

11. Design Patterns (Will)

There are 23 different design patterns but only 6 we need to learn; Examples are numbered and shown below. Patterns 4 &6 have no examples.

11.1. 1. Object Pool

- Pool or reusable objects
- e.g. library book return or connection pools

11.2. 2. Factory

- A class and a series of static methods. Methods in a factory class return an object. E.g. The factory creates a trader;
    - `Trader trader = TheFactory.createATrader();`

11.3. 3. Singleton

- Only one instance of class, good for when you want to use only certain methods of class and not the whole thing; e.g. Math class
- Made up of 3 parts;
    - Private static instance inside class
    - Private no arg constructor (So cant create instance)
    - Returns an instance of the class. If not exist create

11.4. 4. Adapter

- Allow 2 things to work that don't usually work
- E.g. filesystem and printer needs to connect and parse info

11.5. 5. Command

- Like an adapter but split into 4 parts
- Command (thing being passed), receiver (entity that does the work), invoker (doing calling) and client (receives output)

11.6. 6. Observer

- An object (aka subject) maintains a list of its dependents, called observers, and notifies them automatically of any state changes.
- Behaves like an email list, updates are sent to all subscribes and never to others.

1

```
public class ConnectionPool {
    private List<Connection> availableConnections = new ArrayList<Connection>();
    private int howManyConnections=0;

    // use a connection
    public Connection checkout() throws SQLException {
        howManyConnections++;
        return availableConnections.get(howManyConnections-1);
    }
    // return the connection
    public void checkin(Connection c) {
         howManyConnections--;
    }
}
```

2

```
public class TheFactory
{
    public static Trader createATrader()     // Called using the code:
    {    //SuperTrader superTrader = TheFactory.createASuperTrader();
        return new Trader();
    }
    public static SuperTrader createASuperTrader()
```

```
    {
        return new SuperTrader();
    }
```

3

```
public class MySingleton {

    private static MySingleton ms;

    private MySingleton (){
    }

    public static MySingleton getInstance(){

        if(ms == null)
        {
            ms = new MySingleton();
        }
        return ms;
    }
}
```

5

```
// RECEIVER
class Chef {
    public void cookMeal() {
        System.out.println("The meal is ready");
    }
}
// COMMAND
class OrderSlip{
    private Chef chef;

    public OrderSlip(Chef chef) {
        this.chef = chef;
    }
    public void execute() {
        System.out.println("Asking chef to cook meal");
        chef.cookMeal();
    }
}
// INVOKER
class Waiter {
    public void takeInstruction(OrderSlip cmd) {
        System.out.println("Taking instruction");
        cmd.execute();
    }
}
public class RestaurantCustomer {
    public static void main(String[] args) {
        Chef chef = new Chef();
        OrderSlip orderSlip = new OrderSlip(chef);
        Waiter waiter = new Waiter();
        waiter.takeInstruction(orderSlip);
    }
}
```

12. Threads (Ash)

A single sequence of tasks is a thread.

Threads allows a program to **operate more efficiently** by **doing multiple things at the same time.** Threads can be used to perform complicated tasks in the background without interrupting the main program.

12.1. 11.1 Benefits of Multi-threading

- A process split into 2 threads runs quicker than a single threaded process by utilising the idle time of the cores which
- Increases responsiveness
- Performs background and foreground tasks

12.2. 11.2 Making threads

Extending the Thread class: A new class that extends Thread should contain a run() method which overrides that of the parent class. Make an object of this class in the runner.

```
public class MyThread extends Thread {
        public void run() {
        }
}
MyThread myThread = new MyThread();
myThread.start();
```

Implementing the Runnable interface: When implementing Runnable, create an instance of your class that implements Runnable. Insert this object into the constructor of the thread object.

```
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
```

12.3. 11.3 Thread methods

- **start():** alerts the thread scheduler that the thread is ready to compete for processing time.
- **run():** executes the thread.
- **sleep():** pauses the thread for a pre-set amount of time which gives other thread time to work meanwhile.
- **join():** waits for another thread to terminate before it continues.
- **interrupt():** terminates a thread.

These methods prevent a queue from overflowing.

- **wait():** a thread waits to be called upon.
- **notify():** revives a random thread that is waiting.
- **notifyAll():** wakes all the threads.

12.4. 11.4 Multi-threading Hazards and Solutions

Race Condition: The order in which threads execute affects the outcome so there is a different result each time due to the processor switching between threads. Race condition can be prevented by using **locks** or **synchronization**.

Synchronized methods prevent race condition as it allows only 1 method to work on a variable at a time. **Synchronized blocks** are code contained within a method that can only be executed by a single thread at a time. This should be an **atomic operation:** extremely small so as not to hog the resource for a long time.

Re-entrant Locks give a thread different options for proceeding when a resource is being used by another thread. Must use **lock()** and **unlock()** to prevent other methods using at the same time and to allow access once available.

Re-entrant ReadWrite Locks allow access to multiple reading threads but only 1 writing thread at a time. Good for when threads are accessing but not modifying the object.

12.4.1. 11.4.1 Hazards caused by locks and synchronization

Deadlock: Two threads have something the other wants therefore there's no progression. Solve this by ensuring the different threads acquire resources in the same order so are not waiting infinitely for each other.

Livelock: when threads react to each other and are stuck in a continuous loop preventing threads from progressing.

Starvation: A thread cannot proceed due to another thread hogging the resource and not releasing it. Always remember to unlock() after using.

13. Comparisons

There are many different sorting algorithms: quick sort, merge sort, tim sort, insertion sort, selection sort, bubble sort. All sorting algorithms work by comparing pairs of elements. Java chooses the most appropriate algorithm for the collection or array which it is sorting.

However, Java must be told how to compare a pair of objects of a custom class. Comparable and Comparator both **are interfaces and can be used to sort collection elements by defining the sorting order** for custom classes. Below is an example of trying to use `Collections.sort` on objects and as expected, it results in a compile-time error.

Player.java

```
public class Player {

	// Attributes
	private int ranking;
	private String name;
	private int age;

	// Constructor
	public Player(int ranking, String name, int age) {
		this.ranking = ranking;
		this.name = name;
		this.age = age;
	}

	// Getters & Setters
	public int getRanking() {
		return ranking;
	}

	public void setRanking(int ranking) {
		this.ranking = ranking;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public int getAge() {
		return age;
	}
```

```
    public void setAge int age {
        this age = age;
    }

}
```

PlayerSorter.java

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

public class PlayerSorter {

    public static void main(String[] args) {

        // Empty footballTeam list
        List<Player> footballTeam = new ArrayList<>();

        // Creating players (objects)
        Player player1 = new Player(93, "Messi", 34);
        Player player2 = new Player(92, "Ronaldo", 36);
        Player player3 = new Player(90, "van Dijk", 30);
        Player player4 = new Player(91, "Oblak", 28);
        Player player5 = new Player(89, "Ramos", 35);

        // Adding players to 5-a-side footballTeam
        footballTeam.add(player1);
        footballTeam.add(player2);
        footballTeam.add(player3);
        footballTeam.add(player4);
        footballTeam.add(player5);

        // Printing out the footballTeam before sort
        System.out.println("Before sorting: ");
        for (Player player : footballTeam) {

            System.out.println(player.getRanking() + " " +
                                                    player.getName() + " " +
                                                    player.getAge());

        }

        Collections.sort(footballTeam);

        // Printing out the footballTeam after sort
        System.out.println("After sorting: ");
        for (Player player : footballTeam) {

            System.out.println(player.getRanking() + " " +
```

Compile error:
The method sort(List<T>) in the type Collections is not applicable for the arguments (List<Player>)

```
                player.getName() + " " +
                player.getAge());
        }
    }
}
```

13.1. **Comparable**

Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. Comparable affects the original class (the actual class is modified). Comparable provides `compareTo()` method to sort elements.

Using compareTo() method via Comparable interface:

When PlayerSorter.java was run, the footballTeam was sorted from lowest to highest ranking.

Player.java

```
public class Player implements Comparable<Player>{

        .
        .
        .
        .

        @Override
        public int compareTo(Player otherPlayer) {
                return Integer.compare(getRanking(), otherPlayer.getRanking());
        }
}
```

Comparable interface implemented + @Override compareTo() method added to original Player.java

The sorting order is decided by the return value of the `compareTo()` method. The `Integer.compare(x, y)` returns -1 if x is less than y, returns 0 if they're equal, and returns 1 otherwise.

13.2. **Comparator**

The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. Comparator doesn't affect the original class (the actual class is not modified). Comparator provides `compare()` method to sort elements.

Using compare() method via Comparator interface:

PlayerRankingComparator.java

```
import java.util.Comparator;

public class PlayerRankingComparator implements Comparator<Player>{

        @Override
        public int compare(Player firstPlayer, Player secondPlayer) {
                return Integer.compare(firstPlayer.getRanking(),
                                secondPlayer.getRanking());
        }
}
```

PlayerAgeComparator.java:

```
import java.util.Comparator;

public class PlayerRankingComparator implements Comparator<Player>{

	@Override
	public int compare(Player firstPlayer, Player secondPlayer) {
		return Integer.compare(firstPlayer.getRanking(),
				secondPlayer.getRanking());
	}
```

PlayerSorter.java:

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

public class PlayerSorter {

	public static void main(String[] args) {

		// Sort by Ranking (Ascending)
		PlayerRankingComparator playerComparator1 = new
		PlayerRankingComparator();
		Collections.sort(footballTeam, playerComparator1);

		// Sort the SORTED list by Age (Ascending)
		PlayerAgeComparator playerComparator2 = new
		PlayerAgeComparator();
		Collections.sort(footballTeam, playerComparator2);

		// Printing out the footballTeam after sort
		System.out.println("After sorting: ");
		for (Player player : footballTeam) {

		System.out.println(player.getRanking() + " " +
				player.getName() + " " +
				player.getAge());
	}
```