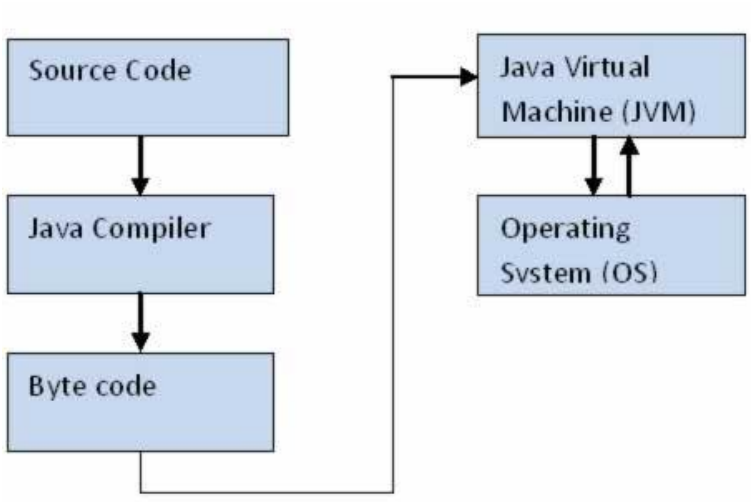
Java Programming Course

Contents

The Environment

Java Architecture



The Java Runtime Environment (JRE) is part of the Java Development Kit, a set of programming tools for developing Java applications. The JRE provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

The JVM interprets compiled Java bytecode into machine code for a computer's processor. Java was designed to allow programs to be built that could run on any platform without having to be rewritten or recompiled by the programmer for each separate platform. A JVM makes this possible because it is aware of the specific instruction lengths and other particularities of the platform.

The JVM Specification defines an abstract machine or processor. The Specification includes an instruction set, a set of registers, a stack, a "garbage heap," and a method area. Once a JVM has been implemented for a given platform, any Java program (which, after compilation, is called bytecode) can run on that platform.

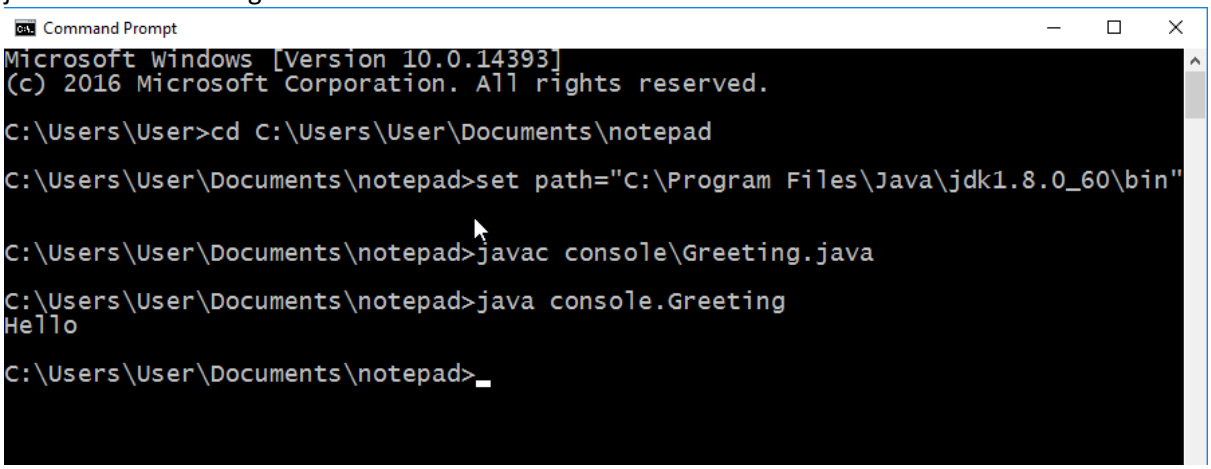
A JVM can either interpret the bytecode one instruction at a time (mapping it to a real processor instruction) or the bytecode can be compiled further for the real processor using what is called a just-in-time compiler.

A simple command line application

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

Compiling and running from the console

set path="C:\Program Files\Java\jdk1.8.0_60\bin"
javac console\Greeting.java
java console.Greeting



http://docs.oracle.com/javase/8/docs/

Jar files

The Java Archive (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications.

Packaging a jar file from the contents of the lib folder and any subdirectories

```
C:\Users\Java\Documents\lib>jar -cf filename.jar *
```

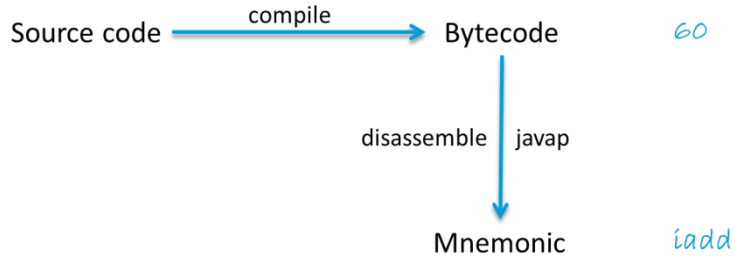
Extracting a jar file

```
C:\Users\Java\Documents\lib>jar -xf filename.jar
```

Launching a jar file

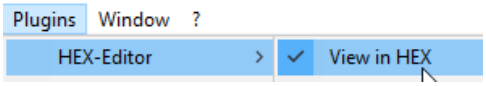
```
C:\Users\Java\Documents\lib>java -jar filename.jar
```

Bytecode



Source code is compiled into bytecode, a sequence of bytes. A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte *opcode* followed by zero or more *operands*. Each bytecode has a corresponding mnemonic.
https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

Notepad++ has a Hex editor plugin



To view mnemonics in IntelliJ, install a plugin
File > Settings > Plugins > install ASM Bytecode Outline
Then right click a .class file and select Show Bytecode Outline

Software used on course

- JDK 8
  - Set JAVA_HOME in user environment variables
  - Path = %JAVA_HOME%\bin
- Git
  - C:\Program Files\Git\bin\git.exe
- Sourcetree
  - https://github.com/javaconsult username client, password tango10
- IntelliJ
- MySQL
  - user1, password
  - root, carpond
- MySQL workbench
- Payara JavaEE server
  - http://www.payara.fish/
  - Derived from GlassFish Server Open Source Edition
  - Unzip, then run `/bin/asadmin start-domain`
  - uninstall.exe -j "%JAVA_HOME%"
- OtrosLogViewer
  - Add glassfish.pattern file to plugins\logimporters
  - click olv.bat to start
  - select “tail glassfish”
- Postman HTTP debugger (Google chrome extension)

- Gradle (optional)
    - Unzip
    - Set GRADLE_HOME in the environment variables
    - add %GRADLE_HOME%/bin to user's PATH environment variable
- Javadoc Documentation paths

File > Project Structure > Platform Settings > SDK

```
JavaSE:                 http://docs.oracle.com/javase/8/docs/api
JavaFX:                 https://docs.oracle.com/javase/8/javafx/api
```

File > Project Structure > Project Settings > Libraries

```
Java EE:                https://docs.oracle.com/javaee/7/api
                        Gradle: org.hibernate.javax.persistence:hibernate-
                                                        jpa-2.1-api:1.0.0.Final
JUnit:                  http://junit.sourceforge.net/javadoc/
```

IntelliJ

https://www.jetbrains.com/help/idea/2016.3/discover-intellij-idea.html

New Project

File > New > Project > Java

File > Project structure > Project settings > Project language level

File > Project Structure > Platform Settings > SDK

```
http://docs.oracle.com/javase/8/docs/api
https://docs.oracle.com/javase/8/javafx/api
```

File > Settings > Editor > General > Show quick documentation on mouse move

Short cuts

The Navigation Bar is a compact alternative to the Project tool window. To access the Navigation Bar, press Alt+Home.

Basic Completion Ctrl+Space

Smart Completion Ctrl+Shift+Space.

Core Language Features

Primitive types

	Bits	Type	Range
boolean	1		true or false
byte	8	integer	-2^7 to 2^7-1
short	16	integer	-2^{15} to $2^{15}-1$
char	16	character	0 to $2^{16}-1$
int	32	integer	-2^{31} to $2^{31}-1$
long	64	integer	-2^{63} to $2^{63}-1$
float	32	floating point	$+/- 3.4 \times 10^{38}$
double	64	floating point	$+/- 1.7 \times 10^{308}$

Java is a *strongly typed* language, meaning that every variable has a type that is known at compile time. Types are divided into two categories: primitive types and reference types. A variable of a primitive type always holds a value of that exact type, while a variable of a class type can hold a reference to an object.

Conversion and casting

```
int i = 5; // assign 5 to i
double d = i; // widening conversion
double x = Math.pow(2, 32);
int y = x;  //narrowing conversion won't compile
int y = (int) x; //cast x as an int
System.out.println(x); // 4.294967296E9
System.out.println(y); // 2147483647
```

Operators

Increment operators

```
int x = 5;
int y = x++; //postfix
System.out.println(y); //5
int z = ++x; //prefix
System.out.println(z); //7
```

instanceof

```
Car car1 = new Car();
System.out.println(car1 instanceof Car); //true
```

Logical operators

```
char c = 'a'; //ascii upper case 65 – 90, lower case 97 - 122
boolean isLowerCase = c >= 97 && c <= 122;
boolean isLetter = c >= 97 && c <= 122 || c >= 65 && c <= 90;

//alternatively, use static methods of the Character class
boolean isLowerCase = Character.isLowerCase(c);
boolean isLetter = Character.isLetter(c);
```

Ternary operator

```
double d = -5.0;
double e = d >=0 ? Math.sqrt(d) : 0;
```

Operator precedence

array [] *object* . *method* () *post* ++ *post* --
pre ++ *pre* -- + - ! ~
cast () new
* / %
+ - *concatenation* +
<< >> >>>
< <= > >= instanceof
== !=
&
^
|
&&
||
?:
= += -= *= /=

Loops and Logic

Conditions

```
double d = -5.0;
if (d>=0) {
	System.out.println(Math.sqrt(d));
}
else {
	System.out.println("complex number");
}
```

For loop

```
for (int i = 0; i < 10; i++) {
	System.out.println(i);
}
```

Break and continue

```
for (int i = 0;; i++) {
	if (i % 2 != 0)
		continue; // starts next iteration
	System.out.println(i);
	if (i >= 100)
		break; // exits current loop
}
```

Labels and nested loops

```
outer: // label
for (int i = 2; i < 100; i++) {
	for (int j = 2; j < i; j++) {
		if (i % j == 0)
			continue outer;
	}
	System.out.println(i);
}
```

Arithmetic series

$$\sum_{k=0}^{10} \mathrm{k} = 0 + 1 + 2 \ldots$$

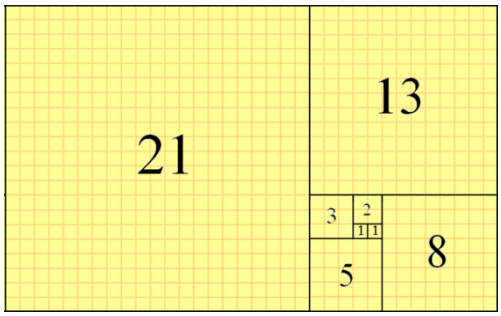
Geometric series

$$\sum_{k=0}^{10} 2^k = 2^0 + 2^1 + 2^2 \ldots$$

double d = Math.pow(2,1);

Fibonacci numbers

A tiling with squares whose side lengths are successive Fibonacci numbers



The sequence Fn of Fibonacci numbers is defined by

$F_n = F_{n-1} + F_{n-2}$

with seed values

$F_1 = 1$, $F_2 = 1$

Using a loop, write the Fibonacci numbers below 100 to the console

0 1 1 2 3 5 8 13 21 34 55 89

Switch block

```
public class Greeting {
    public static void main(String[] args) {
        int number = 0;
        String s = "I";
        switch(s) {
            case "I":
                number = 1;
                break;
            case "V":
                number = 5;
                break;
            case "X":
                number = 10;
                break;
            default:

        }
        System.out.println(number);
    }
}
```

Classes and objects

Defining a class

```
package console;
public class Car {

	//state
	public int speed;
	private int gear;

	//behaviour
	public int getSpeed() {
		return speed;
	}
	public void setSpeed(int speed) {
		this.speed = speed;
	}
	public int getGear() {
		return gear;
	}
	public void setGear(int gear) {
		this.gear = gear;
	}
	public void applyBrake(int decrement) {
		speed -= decrement;
	}
	public void speedUp(int increment) {
		speed += increment;
	}
}
```

Instance variables, accessible throughout the object

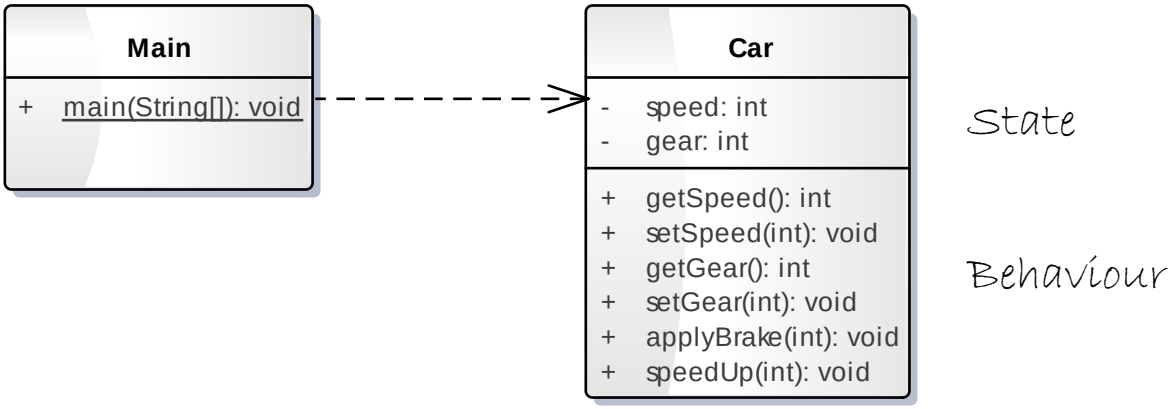
Instantiate a class

```
package console;
public class Main {
	public static void main(String[] args) {
		Car car1 = new Car(); //car1 is a local variable
		car1.speedUp(70);
		car1.applyBrake(20);
		System.out.println(car1.getSpeed());
	}
}
```

Access modifiers

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

UML Class Diagrams



Constructors

```
public class Car {
    //state
    private int speed;
    private int gear;

    public Car() {
    }

    public Car(int speed, int gear) {
        setSpeed(speed);
        setGear(gear);
    }

    // other methods
```

Constructors are used to initialise an object. They're methods with the same name as the class, and don't have a return type. They can be overloaded, meaning additional methods distinguished by their parameters. The expression

```
Car car1 = new Car(50,4);
```

calls the two argument constructor, initialising the object's speed and gear.

Arrays and Strings

Primitive arrays

Arrays store multiple values of a specified type. The following example builds an array object of type int, containing 25 elements. Elements in a numerical array are initialised as zero. A foreach loop iterates through the array.

```
int count=0;
int[] primes = new int[25];
outer: // label
for (int i = 2; i < 100; i++) {
	for (int j = 2; j < i; j++) {
		if (i % j == 0)
			continue outer;
	}
	primes[count++] = i;
}

for (int p : primes) {
	System.out.println(p);
}
```

Shortcut syntax

```
int [] primes = {2, 3, 5, 7, 11};
```

An array of objects

```
Random r = new Random();
int gear = r.nextInt(5)+1; //1 to 5
Car[] cars = new Car[5];
for (int i = 0; i < cars.length; i++) {
	Car car = new Car();
	car.setSpeed(r.nextInt(70));
	car.setGear(r.nextInt(5)+1);
	cars[i] = car;
}
```

Strings

```
String quote = "The unexamined life is not worth living.";

int chars = quote.length(); //40

int index = quote.indexOf("unexamined");  //4

            quote.indexOf("Giraffe");     //-1

String text = quote.substring(4,14);  //unexamined

text = text.toUpperCase(); //UNEXAMINED
```

Mutable and immutable types

Strings are immutable

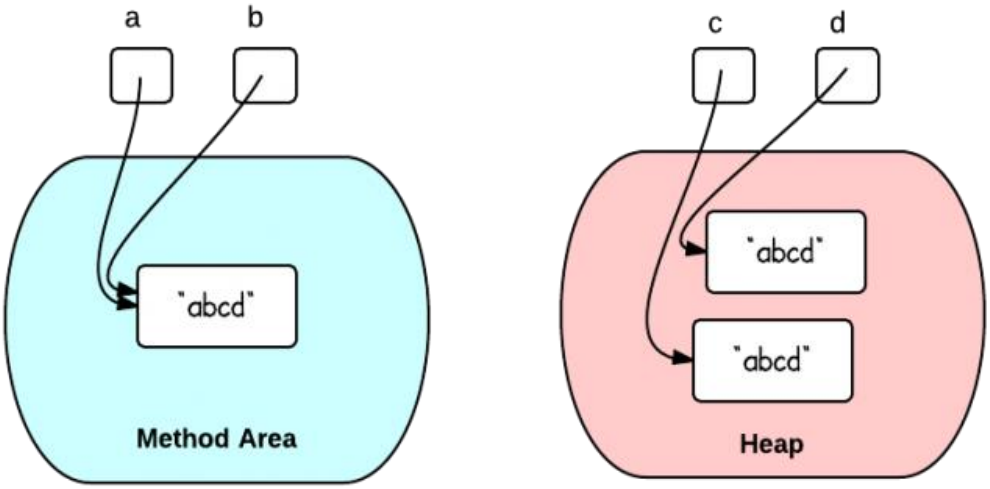
```
String a = "ab";
a = a + "cd"; //new object is created
```

StringBuilder objects are mutable

```
StringBuilder sb1 = new StringBuilder("ab");
sb1.append("cd"); //modifies the same object
```

String constants are interned

```
String a = "abcd";
String b = "abcd";
System.out.println(a == b);  //true – same object

String c = new String("abcd");
String d = new String("abcd");
System.out.println(c == d);  //false -  different objects in the heap
```



A pool of strings, initially empty, is maintained privately by the String class. When the intern method is called, if the pool already contains a string equal to this String object as determined by the equals method, then the string from the pool is returned.

Tweet

- Define a class named Tweet in a package named twitter that includes two private variables; username and text
- Add get and set methods for the two variables
- Add overloaded constructors
- Build an array of five Tweet objects
    - Uneasy lies the head that wears a crown.
    - The fool doth think he is wise, but the wise man knows himself to be a fool.
    - They make a desert and call it peace.
    - What hath God wrought
    - But at my back I always hear time's winged chariot hurrying near
- Iterate through the array, printing the text of each tweet
- Modify the setter method so that tweets above 140 characters are truncated

Enums

An enum is a data type that defines a set of constants. DayOfWeek is an enum in the java.time package.

An enum is defined as follows

```
public enum DayOfWeek {
	MONDAY,
	TUESDAY,
	WEDNESDAY,
	THURSDAY,
	FRIDAY,
	SATURDAY,
	SUNDAY
}
```

Using an enum in a class

```
public class Class1 {
    DayOfWeek day;
}
```

```
Refactor > Encapsulate field
```

```
public class Class1 {
     private DayOfWeek day;

    public DayOfWeek getDay() {
        return day;
    }

    public void setDay(DayOfWeek day) {
        this.day = day;
    }
}
```

```
Class1 c = new Class1();
c.setDay(DayOfWeek.TUESDAY);
```

Using a switch block with an enum

```
	DayOfWeek day = DayOfWeek.MONDAY;

	switch (day) {
	   case MONDAY:
	      break;
	   case TUESDAY     :
	      break;
	   default:
	      break;
	}
```

1. Create an enum named Category with members “CLASSICAL, BIBLICAL, RENAISSANCE, POST_TRUTH”
2. Add a field of type Category to the Tweet class
3. Add public get and set methods for the Category field
4. Set the categories for the Tweet objects created earlier

Keywords

abstract	continue	for	new	switch
assert	default	*goto*[*]	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const[*]	float	native	super	while

Static Methods

Permutations & combinations

```
public class Class1 {
    public static void main(String[] args) {
        double a = Maths.factorial(6); //6 x 5 x 4 x 3 x 2
        double b = Maths.combination(52,4);
        double c = Maths.permutation(52,4);
        double d = Maths.speedOfLight;// public static variable 299 792 458
    }
}
```

$$C_{(n,r)} = \frac{n!}{r!\,(n-r)!}$$

$$P_{(n,r)} = \frac{n!}{(n-r)!}$$

n = set size: the total number of items in the sample

r = subset size: the number of items to be selected from the sample

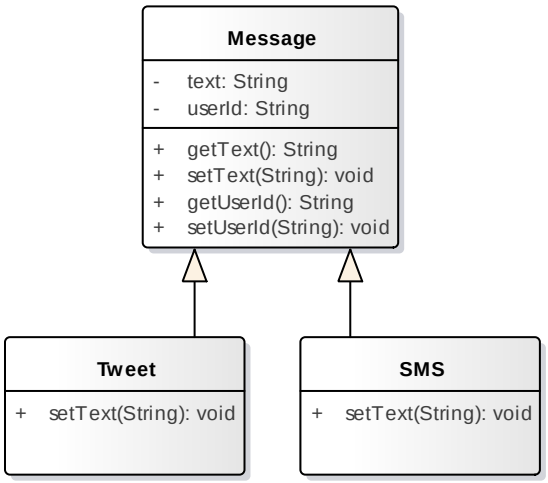
Factorial - The number of ways of arranging n distinct objects into an ordered sequence

Combination - The number of ways to choose a sample of r elements from a set of n distinct objects where order does not matter

Permutation - The number of ways to choose a sample of r elements from a set of n distinct objects where order does matter

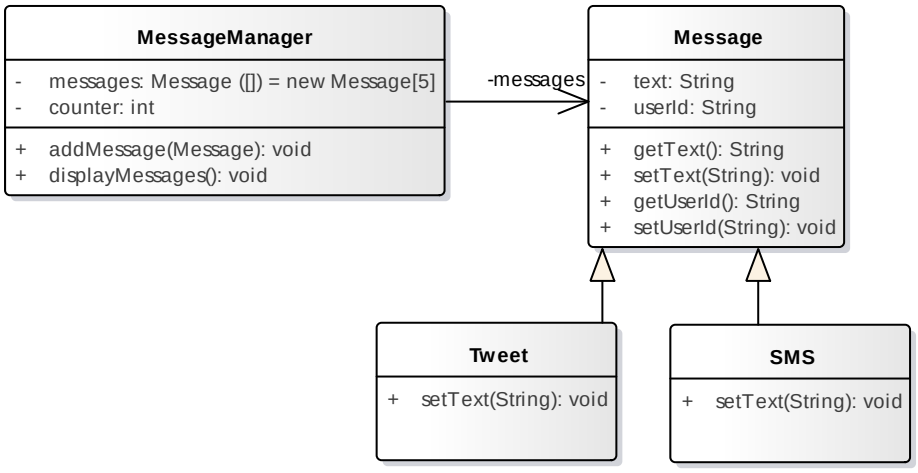
Inheritance

A subclass inherits all non-private members (fields and methods) from its superclass.

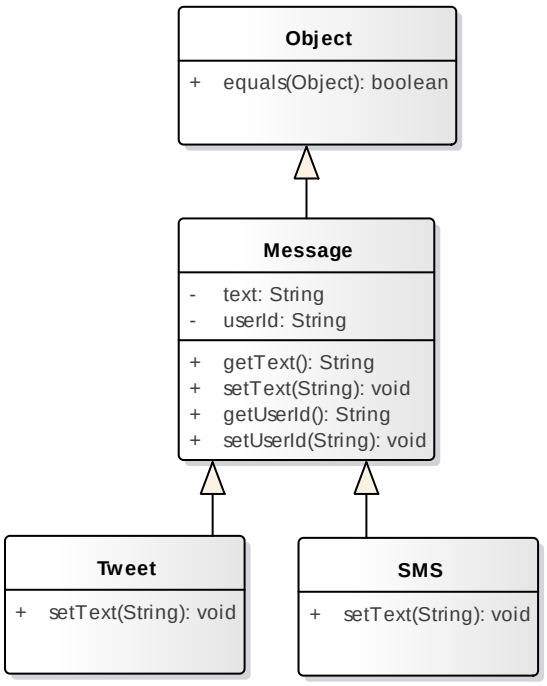


1. Tweet extends Message
2. Generate Message class
3. Refactor > Pull Members Up
4. Override setText method in Tweet class and move 140 character limit to this method
5. Call `super(userId, text)` from Tweet constructor
6. Generate the constructor in the base class
7. Build the SMS class with a 160 character limit
8. Generate the SMS constructor
9. Build a 5 element Message array
10. Add 3 Tweets and 2 SMS objects to the array
11. Iterate through the array, displaying the text of the messages

Associations and generalisations



The Object class



Reference and Value Equality

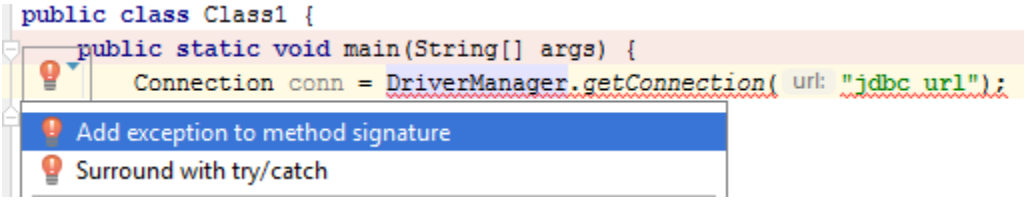
```
public class Class1 {
    public static void main(String[] args) {

        Tweet t1 = new Tweet("user1", "hello");
        Tweet t2 = new Tweet("user1", "hello");
        System.out.println(t1.equals(t2));
    }
}
```

```
public class Message {
    @Override
    public boolean equals(Object obj) {
        return ((Message)obj).text.equals(text) &&
                                    obj.getClass().equals(getClass());
    }
```

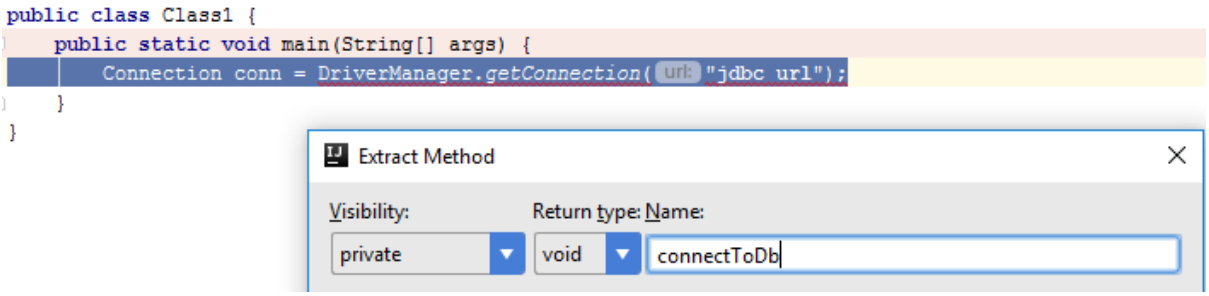
Exceptions

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
A method can either handle an exception, or alternatively throw the exception object down the method call stack.

Extract method

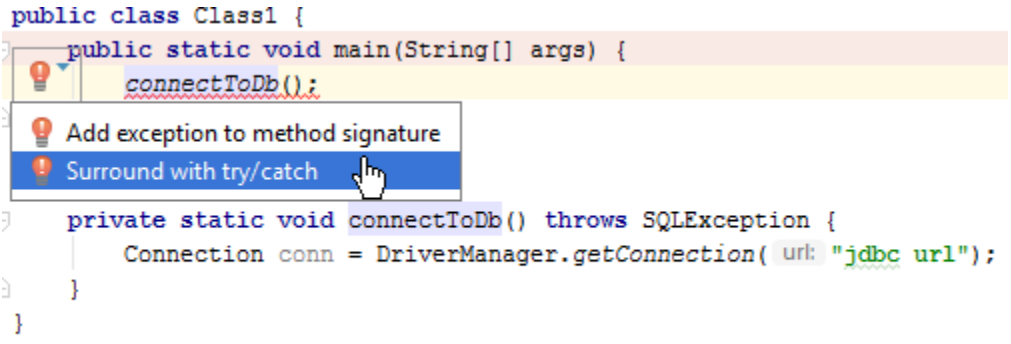


Select Refactor > Extract > Method



Handling exceptions

Surround the connectToDb method call with a try – catch block



Print the exception in the catch block

```
public class Class1 {
    public static void main(String[] args) {
        try {
            connectToDb();
        } catch (SQLException e) {
            System.out.println(e);
        }
    }
```

The method call stack

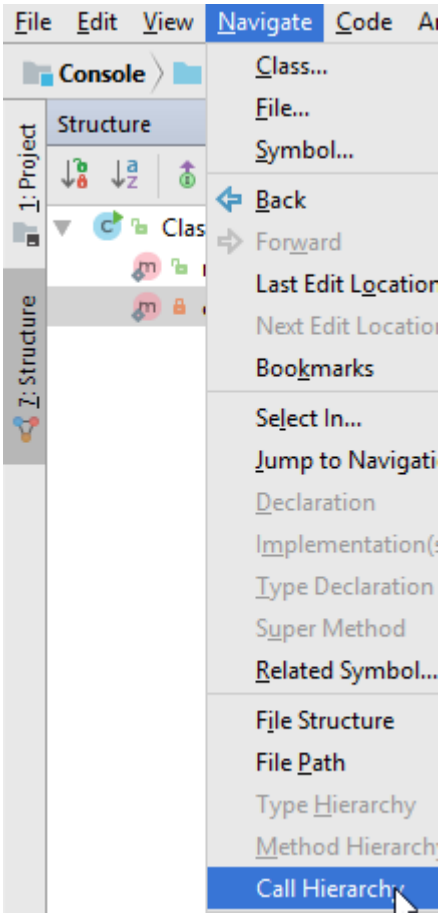
Display the structure View

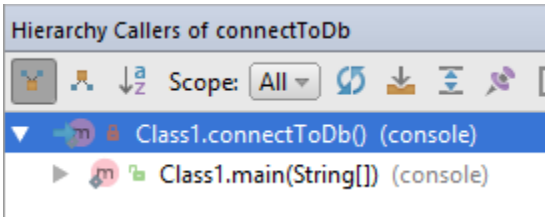
View > Tool Windows > Structure

Display the Call hierarchy

Select the connectToDb method in the structure view

Navigate > Call Hierarchy





Multiple catch blocks

View the JavaDoc for getConnection. Add a second catch block.

```
public static void main(String[] args) {
    try {
        connectToDb();
    }
    catch (SQLTimeoutException e) {
        System.out.println(e);
    }
    catch (SQLException e) {
        System.out.println(e);
    }
}
```

finally blocks

```
private static void connectToDb() throws SQLException {
    Connection conn = null;
    try {
        conn = DriverManager.getConnection("jdbc url");
    }
    finally{
        conn.close();
    }
}
```

Autocloseable objects

```
private static void connectToDb() throws SQLException {
    try(Connection conn = DriverManager.getConnection("jdbc url")){

    }
    catch(SQLException e){
        System.out.println(e);
    }
}
```

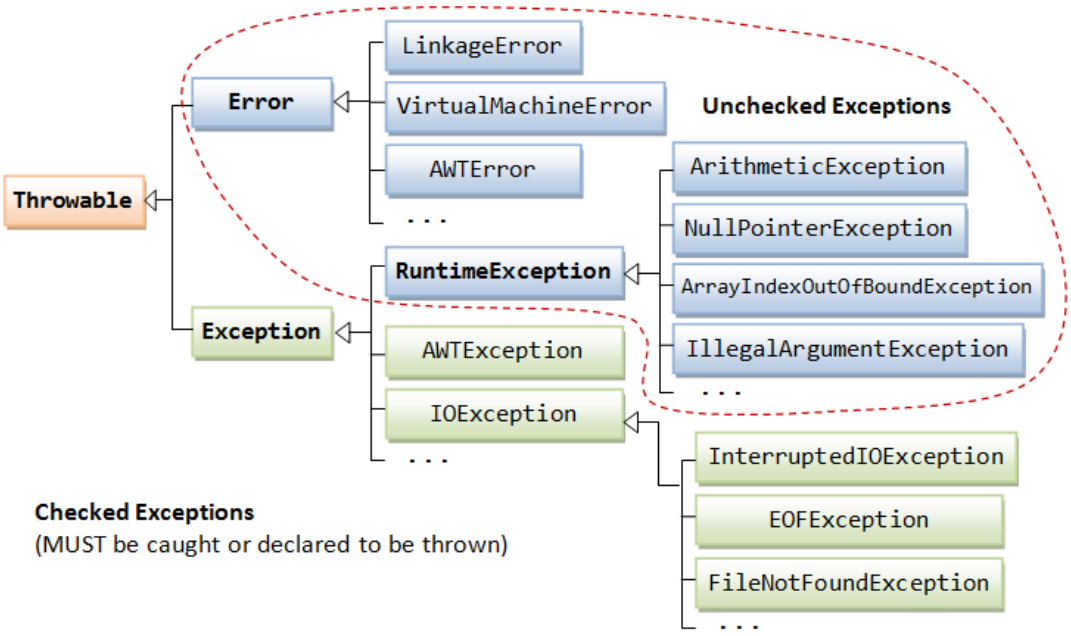
Unchecked exceptions

Dividing an int by zero causes an ArithmeticException

```
try {
    int i = 1/0;
    connectToDb();
}
```

Handling RuntimeExceptions is optional

```
try {
    int i = 1/0;
    connectToDb();
}
catch (ArithmeticException e) {
    System.out.println(e);
}
```



Exception Categories

- Checked exceptions are exceptional conditions that a well-written application should anticipate and recover from.
- Runtime exceptions are exceptional conditions that are internal to the application, and that the application usually can't anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API.
- An error is an exceptional condition that the application usually cannot anticipate or recover from, such as a hardware or system malfunction. An application might choose to catch this exception, in order to notify the user of the problem, or print a stack trace and exit.

Throwing an exception

Use the throw keyword to throw an Exception object from a method

```
if (true) {
   IllegalArgumentException e = new IllegalArgumentException("message...");
   throw e;
}
```

User-defined Exceptions

```
public class MessageFormatException extends RuntimeException {
    public MessageFormatException() {
    }

    public MessageFormatException(String message) {
        super(message);
    }
}
```

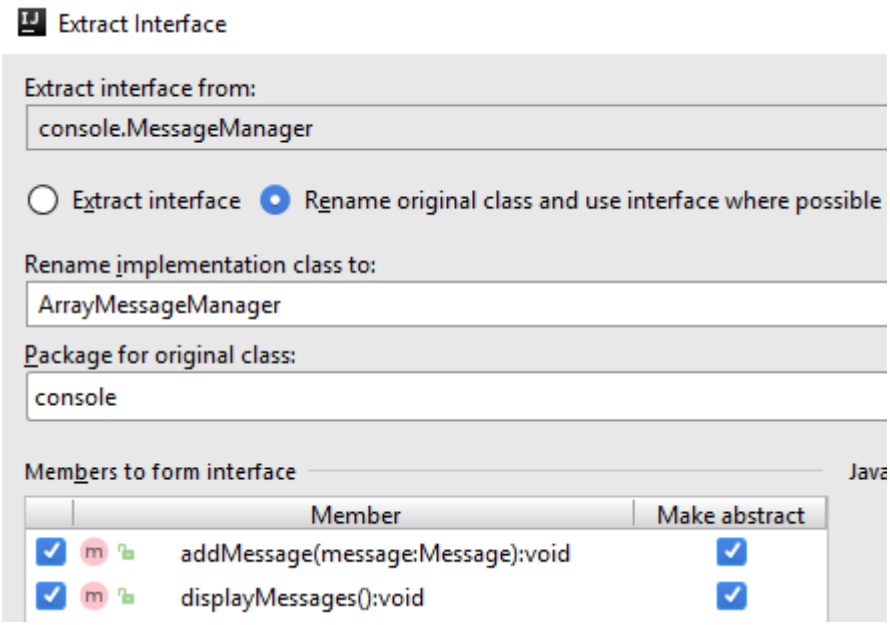
1. Edit the Tweet class's setText property to throw a MessageFormatException if there are more than 140 characters
2. Generate the MessageFormatException class and add two constructors
3. Catch the exception in the main method

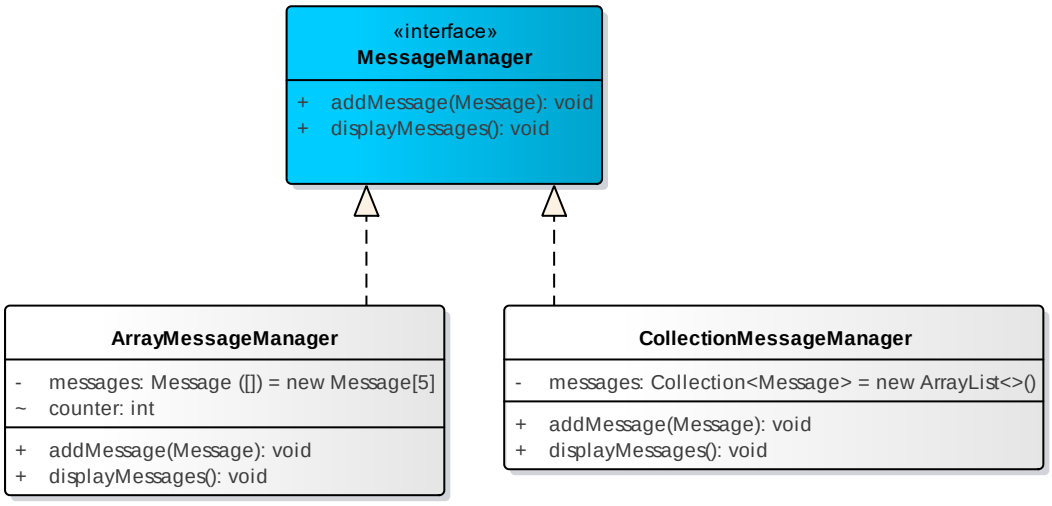
Interfaces

An interface is a specification describing the methods of an object. The implementation of these methods is deferred to a class.

Extract interface

Extract an interface from the MessageManager class and add an implementation





```
public interface MessageManager {
    void addMessage(Message message);

    void displayMessages();
}
```

Collections

Generic class

A generic class is parameterized over types

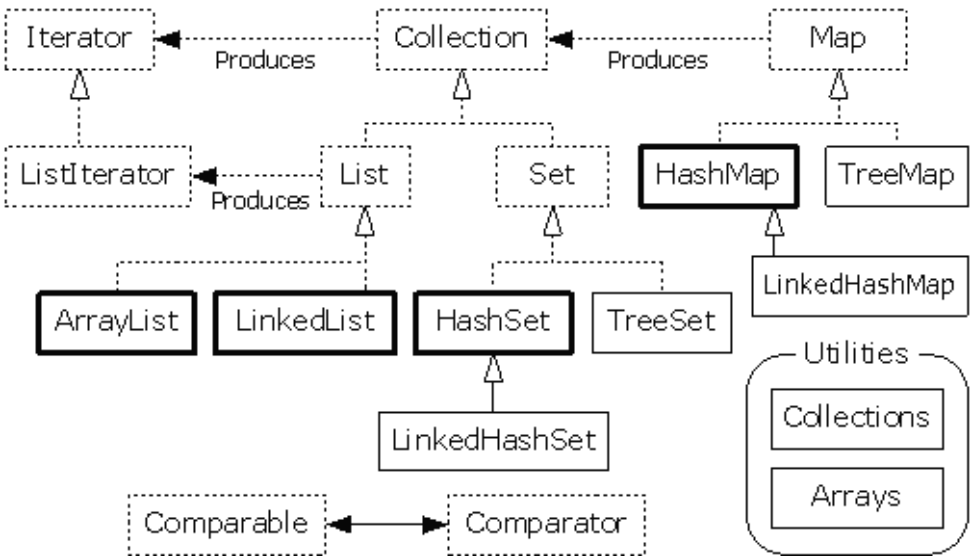
```
package example;

public class MyCollection<E>  {
    private Object[] elementData = new Object[10];
    private int size;
    public boolean add(E e) {
        elementData[size++] = e;
        return true;
    }
}
```

```
Tweet t1 = new Tweet("user1", "Uneasy lies the head that wears a crown");
Tweet t2 = new Tweet("user2", "They make a desert and call it peace");

MyCollection<Message> myCollection = new MyCollection<>();
myCollection.add(t1);
myCollection.add(t2);
```

Type Parameter Naming Conventions

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value

Collections framework



Interfaces in the collection hierarchy

The java.util package contains the collections framework

- Collection is the root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered
- A List is an ordered collection. The user has precise control over where in the list each element is inserted and elements can be accessed by their integer index. Unlike sets, lists typically allow duplicate elements.
- A Set is an unordered collection that contains no duplicate elements. A SortedSet orders its contents.
- A Queue is a FIFO or LIFO collection. A Deque (double ended queue) is a linear collection that supports element insertion and removal at both ends.
- A Map associates unique keys with values. It provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. TreeMap is an ordered implementation of Map, while HashMap is unordered.

ArrayList

```
public class CollectionMessageManager implements MessageManager {

    private Collection<Message>messages = new ArrayList<>();
    @Override
    public void addMessage(Message message) {
        messages.add(message);
    }

    @Override
    public void displayMessages() {
        for (Message message: messages) {
            System.out.println(message.getText());
        }
    }
}
```

Implementing Iterable

```
public class MyCollection<E> implements Iterable<E> {
    private Object[] elementData = new Object[10];
    private int size;
    public boolean add(E e) {
        elementData[size++] = e;
        return true;
    }

    @Override
    public Iterator<E> iterator() {
        return new IteratorImpl<E>();
    }

    @Override
    public void forEach(Consumer<? super E> action) {

    }

    @Override
    public Spliterator<E> spliterator() {
        return null;
    }
```

```
    private class IteratorImpl<T> implements Iterator<T> {
        int count;
        @Override
        public boolean hasNext() {
            return count < size;
        }

        @Override
        public T next() {
            return (T) elementData[count++];
        }
    }
}
```

The Stream API

Lambda Expressions

Functional programming has had a resurgence, due to its applicability to concurrent programming. It can also be usefully applied to manipulating collections.

```
public class LambdaExpressions {
    public static void main(String[] args) {

        Tweet t1 = new Tweet("user1", "Uneasy lies the head that wears a crown");
        Tweet t2 = new Tweet("user1", "The fool doth think he is wise, but the wise
man knows himself to be a fool");
        SMS t3 = new SMS("user2", "What hath God wrought");
        SMS t4 = new SMS("user2", "They make a desert and call it peace");
        SMS t5 = new SMS("user2", "But at my back I always hear time's winged
chariot hurrying near");

        HashSet<Message> messages = new HashSet<>();
        messages.add(t1);
        messages.add(t2);
        messages.add(t3);
        messages.add(t4);
        messages.add(t5);

        List<Message> filteredMessages = //

        for (Message message : filteredMessages) {
            System.out.println(message.getText());
        }
    }
}
```

```
List<Message> filteredMessages = messages.
                                 stream().
                                 filter(m -> m.getText().length() < 50).
                                 collect(Collectors.toList());
```

Functional interfaces

The filter method takes a Predicate argument, which is a functional interface. Functional interfaces have one abstract method; they encapsulate a block of code.

Right click the lambda expression > Refactor > Extract > Variable

```
Predicate<Message> messagePredicate = m -> m.getText().length() < 50;
List<Message> filteredMessages =
          messages.stream().filter(messagePredicate).collect(Collectors.toList());
```

Anonymous inner classes

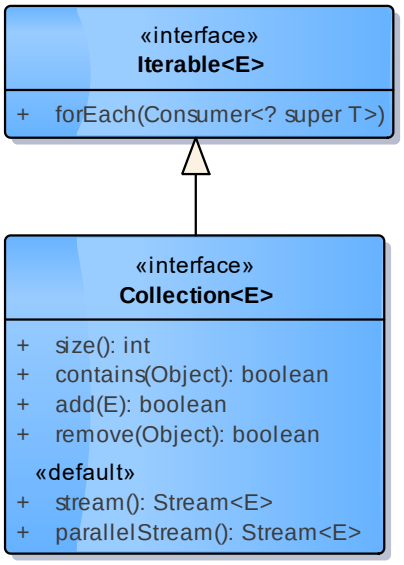
An alternative way of creating an object that implements an interface is to use an anonymous inner class

```
Predicate<Message> messagePredicate = new Predicate<Message>() {
    @Override
    public boolean test(Message message) {
        return false;
    }
};
```

Default methods

Interfaces can include implemented methods, known as default methods. <? super T> is a lower bounded wildcard, restricting the unknown type to be a specific type or a super type of that type.

```
 public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action) {
        //implementation
    }
}
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean contains(Object o);
    boolean add(E e);
    boolean remove(Object o);

    default Stream<E> stream() {
        //implementation
    }
    default Stream<E> parallelStream() {
       //implementation
    }
}
```



Consumer is a functional interface that takes a generic argument and has a void return type

```
filteredMessages.forEach(m -> System.out.println(m.getText()));
```

Collecting results

The abstract stream can be “collected” into a result. Supplier and BiConsumer are functional interfaces that can describe the creation and population or a result container, in this example an ArrayList.

```
//supplier - function that creates a new result container
Supplier<ArrayList<Message>> supplier = () -> new ArrayList<Message>();

//accumulator - function for incorporating an additional element into a result
BiConsumer<ArrayList<Message>, Message> accumulator = (x, y) -> {
    x.add(y);
};

//combiner - function for combining two values, compatible with the accumulator
BiConsumer<ArrayList<Message>, ArrayList<Message>> combiner =
        (x, y) -> {
```

```
            x.addAll(y);
        };
```

Chaining methods

```
List<String> filteredMessages = messages.
                                stream().
                                filter(m -> m.getText().length() < 50).
                                map(m->m.getText()).
                                collect(Collectors.toList());
```

Method references

Method references are compact lambda expressions for methods that already have a name.

```
filteredMessages.forEach(System.out::println);
```

There are four kinds of method references:

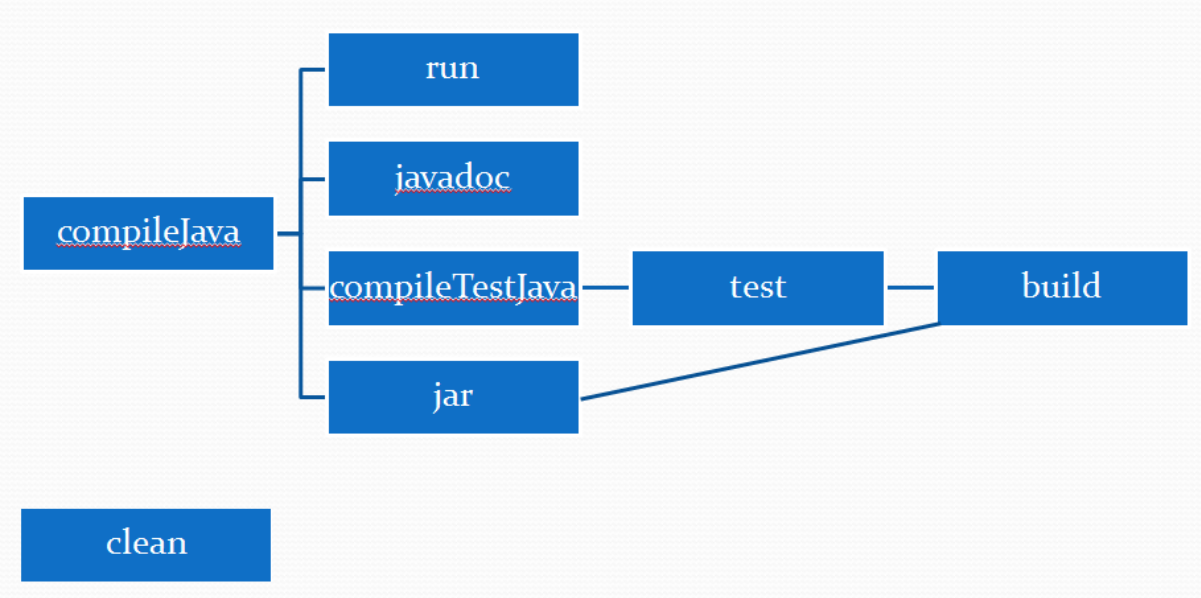
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

Gradle

Gradle is an open source build automation system. It can automate building, testing, publishing and deployment. Plugins are a mechanism to extend core Gradle with additional functionality. Edit build.gradle, adding the application plugin, setting mainClassName to the package qualified name of the class containing the main method and changing sourceCompatibility to 1.8.

```
apply plugin: 'java'
apply plugin:'application'
mainClassName = "com.example.Class1"

sourceCompatibility = 1.8

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

To run a Gradle task, select Window > Show View > Other > Gradle > Gradle Tasks. The application plugin has a **run task** that starts the application by calling the main method, configured in build.gradle as the mainClassName property. The run task depends on the classes task, defined in the Java plugin. the Eclipse plugin generates a .project file



Abbreviated view of dependencies between Gradle tasks in the Java and Application plugins

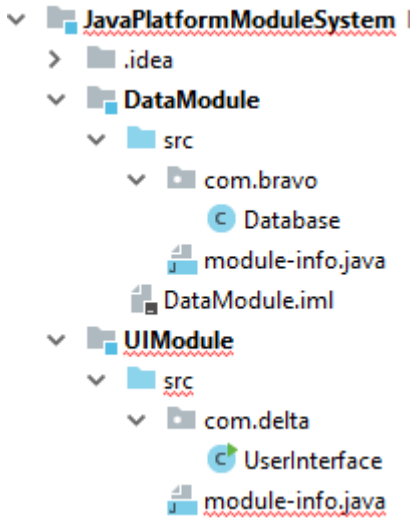
View list of tasks with >Gradle tasks

Directories

src/main/java	Application sources
src/main/resources	Application resources
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
build	Output
build.gradle	Build script

The Java Platform module system

Prior to Java 9, access modifiers can be used to encapsulate classes, but packages couldn't be encapsulated. A module is a collection of packages. The module-info.java file *requires* dependent modules and *exports* packages. JARs without a module-info file are automatic, meaning all modules are required and all packages are exported. Every module requires the java.base module by default. Modules generally have the name of the root package that they contain, for example java.sql.



- Create an IntelliJ Java project and add two modules to it: DataModule and UIModule
- Add a dependency on DataModule to UIModule
    - File > Project Structure
    - Select UIModule
    - Click the + button and add a dependency on DataModule

IntelliJ DataModule

```
package com.bravo;
import java.sql.Connection;
public class Database {
    public Connection getConnection(){
        Connection c = null;
        return c;
    }
}
```

Add a file named module-info.java to the package root. Use the *requires* keyword to indicate a dependency on a module. A dependency on java.base is implied
https://docs.oracle.com/javase/9/docs/api/java.base-summary.html

The exports keyword makes a package available to another module.

```
module com.bravo{
    //requires java.base //every module requires the java.base module by default
    requires java.sql; //declares a dependency on this module
    exports com.bravo; //this package is accessible outside the module
}
```

IntelliJ UIModule

```
package com.delta;
import com.bravo.Database;
public class UserInterface extends Application {
    private Database database = new Database();
    public void start(Stage stage) {
        Connection c = database.getConnection();
        GridPane pane = new GridPane();
        Scene scene = new Scene(pane, 400, 300);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

The Application class is in the javafx.graphics module, the Connection interface is in the java.sql module and the Database class built earlier is in the com.bravo module. These modules are added as dependencies. When running the application, the JavaFX class need to access the package containing the UI code, so it is exported.

```
module com.delta{
    requires java.sql;
    requires javafx.graphics;
    requires com.bravo;

    exports com.delta; //this package must be accessible to JavaFX classes
}
```

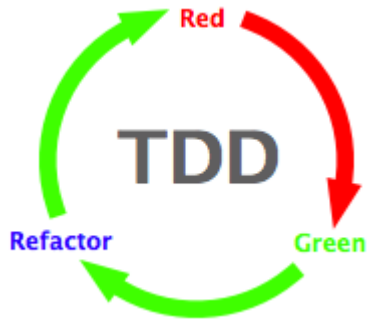
Instead of requiring java.sql in the com.delta module, it can be declared as a transitive dependency in the com.bravo module

```
module com.bravo{
    requires transitive java.sql;//refers to a module
    exports com.bravo; //refers to a package
}

module com.delta{
    //requires java.base; //this is implied
    //requires java.sql; //transitive dependency in com.bravo
    requires javafx.graphics;
    requires com.bravo;
    exports com.delta;
}
```

The java.sql module requires java.logging and java.xml
The javafx.graphics module requires transitive javafx.base and exports multiple packages including javafx.scene.layout

Test-driven development

Overview



TDD is a software development process that relies on the repetition of a very short development cycle

1. Write an (initially failing) automated test case that defines a desired improvement or new function
2. Produce the minimum amount of code to pass that test
3. Refactor the new code to acceptable standards

Benefits

- Test cases force the developer to consider how functionality is used by clients, focussing on the interface before the implementation
- Helps to catch defects early in the development cycle
- Requires developers to think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces.
- Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Unit tests.

- Single classes
- replace real collaborators with test doubles
- ensure high quality code

Integration tests.

- the code under test is not isolated
- run more slowly than unit tests
- verify that modules are cooperating effectively
- Integration testing is similar to unit testing in that tests invoke methods of application classes in a unit testing framework. However, **integration tests do not use mock objects to substitute implementations for service dependencies**. Instead, integration tests rely on the application's services and components. The goal of integration tests is to exercise the functionality of the application in its normal run-time environment.

Acceptance tests.

- multiple steps that represent realistic usage scenarios of the application as a whole.
- scope includes usability, functional correctness, and performance.

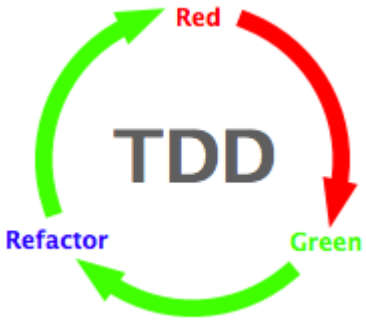
Phases when writing a test

1. Arrange – create objects
2. Act – execute methods to be tested
3. Assert – verify results

```
package entity;
import static org.junit.Assert.*;
import java.time.LocalDate;
import org.junit.Test;

public class FilmTest {
	//The Test annotation tells JUnit that the public void method to which it
	//is attached can be run as a test case.
	@Test
	public void constructorShouldInitialiseFields() {
		//arrange and act
		Film film = new Film("The Pink Panther", 5,
					LocalDate.of(1964, 1, 20), Genre.COMEDY);
		Long id = film.getId();
		String title = film.getTitle();
		int stock = film.getStock();
		LocalDate released = film.getReleased();
		Genre genre =  film.getGenre();

		//assert
		assertNull(id);
		assertEquals("The Pink Panther", title);
		assertEquals(5, stock);
		assertEquals(LocalDate.of(1964, 1, 20), released);
		assertEquals(Genre.COMEDY, genre);
	}
```

TDD Rhythm

1. write a test that fails (RED)
2. make the code work (GREEN)
3. rewrite code so that it's maintainable (REFACTOR)
   - KIS (keep it simple) writing the smallest amount of code to make the test pass leads to simple solutions.
   - Avoid unnecessary methods YAGNI "You aren't going to need it"
   - DRY (don't repeat yourself)
   - SRP (single responsibility principle)
   - add Javadocs



Apply this to the above example

1. Add the FilmTest class to the src/test/java folder.
   Generate the Film class and Genre enum in the src/main/java directory, using Eclipse. Add a constructor, fields, get and set methods.
   Run >gradle test, expecting the assertions to fail.
2. Complete the methods, so that the assertions pass.
3. Refactor the code, for example separate fields from methods in the code so that it's easier to read. Then commit the changes to Version Control.

Unit tests and expected exceptions

The Test annotation can take an *expected* attribute, indicating the type of exception that a method is expected to throw.

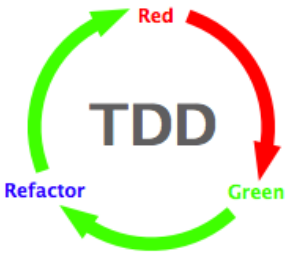
```
@Test(expected = IllegalArgumentException.class)
public void constructorShouldThrowExceptionIfStockNegative() {
	new Film("The Pink Panther", -1, LocalDate.of(1964, 1, 20), Genre.COMEDY);
}
```

Following the TDD rhythm

1. Add the above method to the FilmTest class
   Run >gradle test, expecting the test to fail.
2. Amend the constructor so that the test passes
3. Refactor the code, for example it might be preferable to throw the exception from the setStock method rather than directly from the constructor

Override equals

Add the following method to the FilmTest class in the FilmStore project



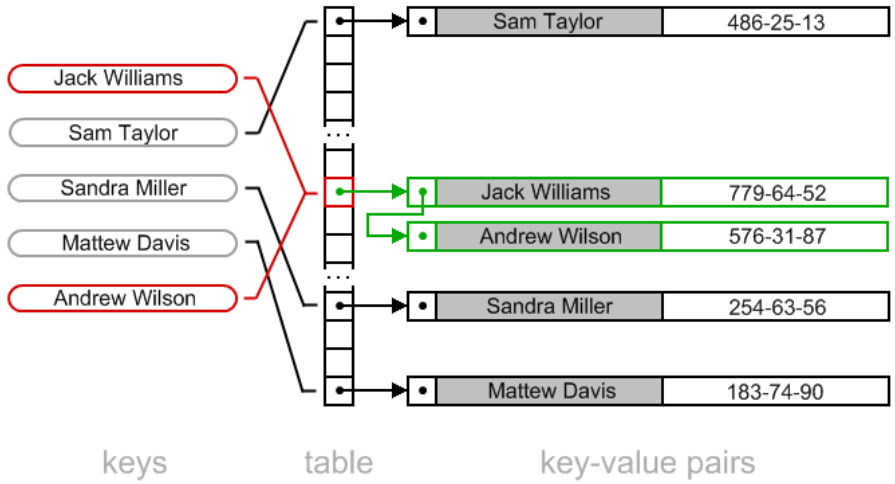
1. run the test; it should fail
2. override equals method in the object class so that the assertion passes
3. refactor

```
@Test
public void filmShouldWorkWithList() {
    // arrange
    Film film1 = new Film("The Pink Panther", 5,
                                    LocalDate.of(1964, 1, 20), Genre.COMEDY);
    Film film2 = new Film("The Pink Panther", 5,
                                    LocalDate.of(1964, 1, 20), Genre.COMEDY);
    List<Film> films = new ArrayList<>();
    //act
    films.add(film1);
    films.remove(film2);
    //assert
    assertEquals(0,films.size());
}


@Test
public void filmsWithSameTitleShouldBeEqual() {
    // arrange
    Film film1 = new Film();
    film1.setTitle("The Godfather");
    Film film2 = new Film();
    film2.setTitle("The Godfather");

    // act (execute methods under test) and assert (verify test results)
    assertTrue(film1.equals(film2));
}
```

Override hashCode

A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

Java uses a strategy called chaining, in which each slot of the array contains a link to a singly-linked list containing key-value pairs with the same hash. New key-value pairs are added to the end of the list. Lookup algorithm searches through the list to find matching key. Initially table slots contain nulls. The value returned by hashCode() is the object's hash code, which is the object's memory address in hexadecimal. If two objects are equal, their hash code must also be equal.




```
@Test
public void filmShouldWorkWithSet() {
    // arrange
    Film film1 = new Film("The Pink Panther", 5,
                                            LocalDate.of(1964, 1, 20), Genre.COMEDY);
    Film film2 = new Film("The Pink Panther", 5,
                                            LocalDate.of(1964, 1, 20), Genre.COMEDY);
    Set<Film> films = new HashSet<>();
    //act
    films.add(film1);
    films.remove(film2);
    //assert
    assertEquals(0,films.size());
}
```

```
@Test
public void filmsWithSameTitleShouldHaveEqualHashcodes() {
    // arrange
    Film film1 = new Film("The Pink Panther", 5,
                                            LocalDate.of(1964, 1, 20), Genre.COMEDY);
    Film film2 = new Film("The Pink Panther", 5,
                                            LocalDate.of(1964, 1, 20), Genre.COMEDY);
    assertTrue(film1.hashCode() == film2.hashCode());
}
```

String formatting

The toString method of the Film class could also be overridden to return a String representation of the object. This is an example, using the variable parameter format method of the String class to display something like "`The Pink Panther, stock 5, was released in January 1964`"

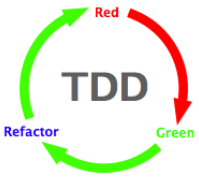
```
@Override
public String toString() {
	return String.format("%s, stock %d, was released in %tB %3$tY",
				title, stock, getReleased());
}
```

Collections classes

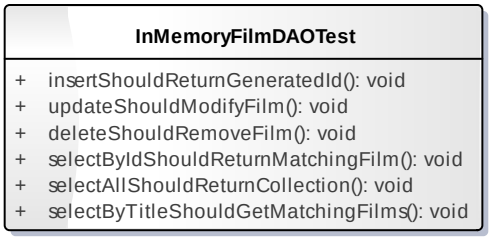
```
public class CollectionTest {

	Film film1 = new Film("The Godfather", 2, LocalDate.of(1972, 4, 17), Genre.CRIME);
	Film film2 = new Film("The Godfather", 2, LocalDate.of(1972, 4, 17), Genre.CRIME);

	@Test
	public void listCanStoreDuplicates() {
		//arrange
		List<Film> set = new ArrayList<>();
		//act
		boolean film1Added = set.add(film1);
		boolean film2Added = set.add(film2);
		//assert
		assertTrue(film1Added);
		assertTrue(film2Added);
		assertEquals(2,set.size());
	}

	@Test
	public void setContainsUniqueObjects() {
		//arrange
		Set<Film> set = new HashSet<>();
		//act
		boolean film1Added = set.add(film1);
		boolean film2Added = set.add(film2);
		//assert
		assertTrue(film1Added);
		assertFalse(film2Added);
		assertEquals(1,set.size());
	}

	@Test
	public void mapContainsUniqueKeys() {
		//arrange
		Map<Long, Film> map = new HashMap<>();
		//act
		Film previousValue1 = map.put(1L, film1);
		Film previousValue2 = map.put(1L, film2);
		//assert
		assertNull(previousValue1);
		assertEquals(film1, previousValue2);
		assertEquals(1,map.size());
	}

	@Test
	public void addUpdateAndRemoveFromMap() {
		//arrange
		Map<Long, Film> map = new HashMap<>();
```

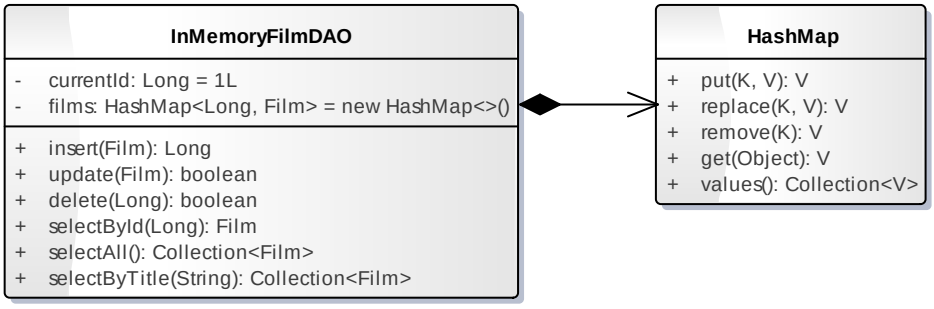
```
            //act
            Film previousValue1 = map.put(1L, film1); //add key and value to a map
            Film previousValue2 = map.replace(1L, film2); //null if key isn't in map
            Film removedFilm = map.remove(1L); //remove value with specified key
            //assert
            assertNull(previousValue1);
            assertEquals(film1, previousValue2);
            assertEquals(film2, removedFilm);
            assertTrue(map.isEmpty());
        }
}
```

Data Access Object



A Data Access Object (DAO) abstracts and encapsulates access to the data source. In this example, the data is stored in memory, as a Map. Using TDD, generate the InMemoryFilmDAO class in a package named session, in the src/java/main folder.





The HashMap class contains methods that could be used to implement the methods of the InMemoryFilmDAO class. For example the insert method could be implemented using the HashMap's put method.

InMemoryFilmDAO

Complete the selectByTitle method, using a lambda expression.

```
public class InMemoryFilmDAO {
        private Long currentId = 1L;
```

```
private HashMap<Long, Film> films = new HashMap<>();
public Collection<Film> selectByTitle(String search) {
       Collection<Film> filmCollection = films.values();
```

Executing Streams in Parallel

```
public static long primeCount(int limit) {
   long count =
            IntStream.range(2, limit).
            parallel().
            filter(p->!IntStream.range(2, p).anyMatch(n -> p % n ==0)).
            count();
   return count;
}

//replacing (2, p) with (2, (int)Math.sqrt(p)+1) significantly improves
performance
```

Parallel tests

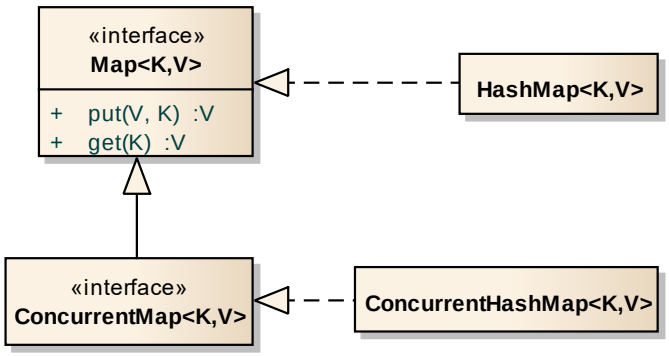
The tempus-fugit library offers a RepeatingRule and ConcurrentRule that can be used to run a test method multiple times and across multiple threads. This can be useful when writing load tests.

```
public class ParallelTest {
    private static Map<Long, Film> films = new HashMap<>();
    //private static Map<Long, Film> films =
                                    new ConcurrentHashMap<>();

    private static InMemoryFilmDAO sut =
                                   new InMemoryFilmDAO(films, null);

    @Rule
    public ConcurrentRule concurrently = new ConcurrentRule();
    @Rule
    public RepeatingRule repeatedly = new RepeatingRule();

    @Test
    @Repeating(repetition = 1000) //execute method 1000 times
    @Concurrent(count = 5) //in 5 threads
    public void insertFilmsConcurrently() {
        sut.insert(new Film());
    }

    @AfterClass
    public static void numberOfFilmsInserted() {
        assertEquals(5000, sut.selectAll().size());
    }
}
```

This requires the following dependency

```
dependencies {
    testCompile 'com.google.code.tempus-fugit:tempus-fugit:1.1'
```

java.util.concurrent



ConcurrentHashMap is a hash table supporting full concurrency of retrievals and high expected concurrency for updates.

Collection classes in the java.util package aren't synchronised. Use the Collections class to obtain a thread safe collection:

```
List list = Collections.synchronizedList(new ArrayList());
```

or use a Collection class in the java.util.concurrent package.

Update the counter and Map fields in the `InMemoryFilmDAO` class so that the assertions pass.

Thread interference

Source code

```
public void deposit(int amount) {
     balance += amount;
}
```

Corresponding mnemonic, bytecode and description

```
 0  aload_0 [this]     [2A] push this onto the stack
 1  dup                [59] duplicate the value on top of the stack
 2  getfield threads.bank.Account.balance : int [18]
                       [B4] get the value of the instance variable
 5  iload_1 [amount]   [1B] push int value from local variable 1 onto stack
 6  iadd               [60] add two ints
 7  putfield threads.bank.Account.balance : int [18]
                       [B5] set field to value
10  return             [B1] return void from method
```

Viewing Account.class in a binary editor

```
00000200  00 06 00 01 00 01 00 17  00 15 00 01 00 09 00 00
00000210  00 43 00 03 00 02 00 00  00 0B 2A 59 B4 00 12 1E
00000220  64 B5 00 12 B1 00 00 00  02 00 0C 00 00 00 0A 00
00000230  02 00 00 00 56 00 0A 00  57 00 0D 00 00 00 16 00
```

java.util.concurrent.atomic

This package is a small toolkit of classes that support lock-free thread-safe programming on single variables. The assertion in the `ParallelTest` class may fail if a primitive such as a long is used to generate the next id, as operations such as ++ are not atomic. Instead, use the incrementAndGet() method of AtomicLong.

Synchronized methods

Synchronizing the insert method will prevent multiple threads executing it simultaneously. The lock is held by the current object

```
public class InMemoryFilmDAO implements FilmDAO {
    @Override
    public synchronized Long insert(Film film) {
```

Dates and Times

Instant

```
// An instant represents a point in time
// the origin is set arbitrarily at 1 Jan 1970 GMT
// see console. DatesAndTimes
System.out.printf("Instant now in seconds %d%n",Instant.now().getEpochSecond());
Instant start = Instant.now();
Thread.sleep(1000);
Instant end = Instant.now();
//a duration is the amount of time between two instants
System.out.printf("Duration %d%n",
                          Duration.between(start, end).toMillis());
```

LocalDate

```
// A LocalDate has no time zone information
LocalDate today = LocalDate.now();
LocalDate date1 = LocalDate.of(2015, 1, 20);
LocalDate date2 = date1.plusMonths(1);
System.out.printf("LocalDate %s%n",date2);
LocalTime time1 = LocalTime.now();
LocalTime time2 = LocalTime.of(12, 30);
System.out.printf("LocalTime %s%n",time2);
LocalDateTime localDateTime1 = LocalDateTime.of(2014,3,29,14,45);
LocalDateTime localDateTime2 = localDateTime1.plusHours(24);
System.out.printf("LocalDateTime %s%n",localDateTime2);
```

ZonedDateTime

```
LocalDateTime localDateTime1 = LocalDateTime.of(2014,3,29,14,45);
ZonedDateTime zonedTime1 = ZonedDateTime.of(localDateTime1,
                                        ZoneId.of("Europe/Berlin"));
System.out.printf("ZonedDateTime %s%n",zonedTime1);
ZonedDateTime zonedTime2 = zonedTime1.plusHours(24);
System.out.printf("Summer time %s%n",zonedTime2);
```

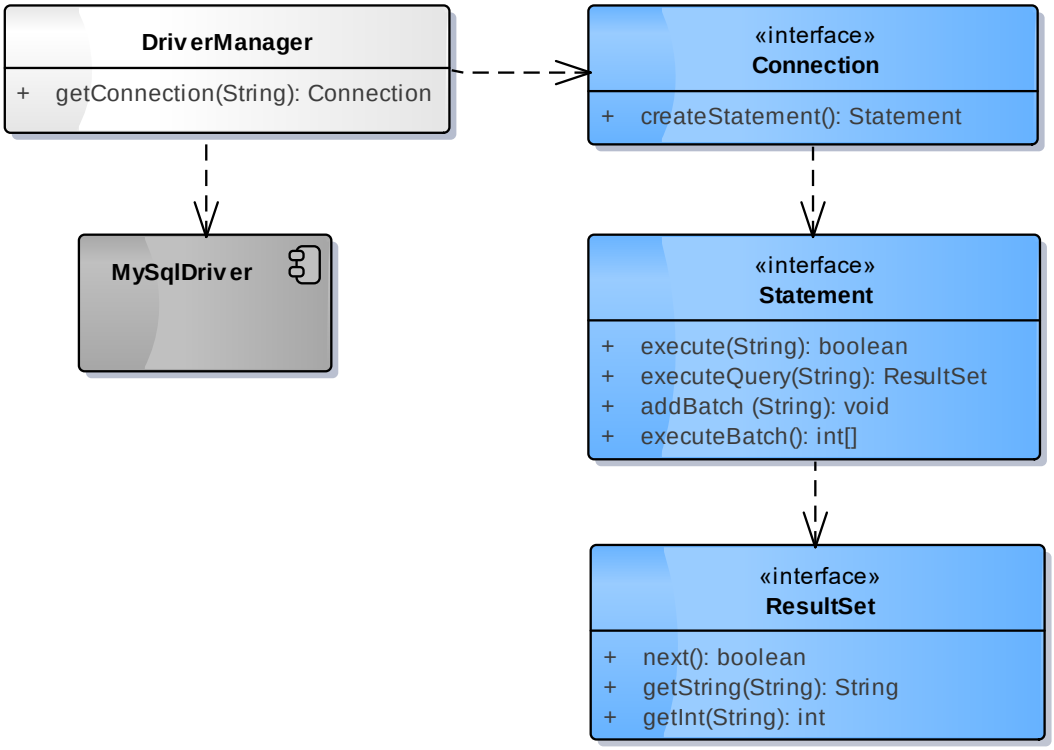
DateTimeFormatter

```
DateTimeFormatter formatter =
           DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL);
System.out.printf("Formatted: %s%n", formatter.format(zonedTime2));
```

Databases

JDBC

Java DataBase Connectivity is a programming interface that abstracts interaction with a database



SQLite

- embedded SQL database engine
- does not have a separate server process; reads and writes directly to a file

```
public static void main(String[] args) {
    String url = "jdbc:sqlite:C:/Users/User/Downloads/sqlitedb.db";
    try (Connection conn = DriverManager.getConnection(url);){
        System.out.println("Connection to SQLite has been established.");
        Statement statement = conn.createStatement();
        ResultSet resultSet = statement.executeQuery("select * from film");
        while(resultSet.next()){
            System.out.println(resultSet.getString("title"));
        }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
```

Rather than writing a separate unit test method for each operation (insert, read, update, delete), it can be easier to test all 4 operations inside the same test method.

```
public class JdbcFilmDAOIT {

    @Before
    public void init(){
        //truncate table
    }

    @Test
    public void test() {
        //arrange
        Film film1 = new Film("The Pink Panther", 5,
                                LocalDate.of(1964, 1, 20), Genre.COMEDY);
        Film film2 = new Film("The Godfather", 2,
                                LocalDate.of(1972, 4, 17), Genre.CRIME);
        Film film3 = new Film("Avatar", 2,
                                LocalDate.of(2009, 7, 2), Genre.SCIENCE_FICTION);
         FilmDAO dao = new JdbcFilmDAO();

         //act and assert
         long id1 = dao.insert(film1);
         long id2 = dao.insert(film2);
         long id3 = dao.insert(film3);

         assertEquals(film1, dao.selectById(id1));
         assertEquals(film2, dao.selectById(id2));
         assertEquals(3, dao.selectAll().size());
         assertEquals(2, dao.selectByTitle("at").size());

         film1.setTitle("The Return of the Pink Panther");
         dao.update(film1);

         assertEquals("The Return of the Pink Panther",
         dao.selectById(id1).getTitle());

         dao.delete(id1);
         dao.delete(id2);
         dao.delete(id3);
         assertEquals(0,dao.selectAll().size());
    }
}
```

Pessimistic concurrency

Transaction Isolation Levels

Transaction isolation level	Dirty reads	Nonrepeatable reads	Phantoms
READ UNCOMMITTED	X	X	X
READ COMMITTED		X	X
REPEATABLE READ (default)			X
SERIALIZABLE			

Dirty Reads	a transaction reads data that has not yet been committed
Nonrepeatable	a transaction reads the same row twice but gets different data each time
Phantoms	a row that matches the search criteria but is not initially seen

```
try (Connection connection = DriverManager.getConnection(url, username, password);
     PreparedStatement statement = connection.prepareStatement(sql)) {
    connection.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
    connection.setAutoCommit(false);
    //CRUD operations
    connection.commit();
} catch (Exception ex) {
    throw new FilmException(ex.getMessage());
}
```

Optimistic concurrency

Optimistic concurrency control assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read.

The SQL update for the Film table could be as follows:

```
String sql = "update film set title = ?, released = ?, genre = ?, stock = ?,
                          version = version + 1 where id = ? and version = ?";
```

Add a version field of type int to the Film class

Partial implementation

```
public class JdbcFilmDAO implements FilmDAO {

    private String url = "jdbc:mysql://localhost:3306/filmstore
                                ?verifyServerCertificate=false&useSSL=true";
    private String username = "root";
    private String password = "carpond";

    @Override
    public Long insert(Film film) {
        String sql = "insert into Film (title, stock, released, genre, version)
                                                                values (?,?,?,?,?)";
        try (Connection connection = DriverManager.getConnection(
                                                        url, username, password);
             PreparedStatement statement = connection.prepareStatement(
                                sql, Statement.RETURN_GENERATED_KEYS)) {
            statement.setString(1, film.getTitle());
            statement.setInt(2, film.getStock());
            LocalDate released = film.getReleased();
            statement.setDate(3, Date.valueOf(released));
            statement.setString(4, film.getGenre().toString());
            statement.setInt(5, 0);
            statement.execute();
            ResultSet rs = statement.getGeneratedKeys();
            if (rs.next()) {
                Long generatedId = rs.getLong(1);
                film.setId(generatedId);
                return generatedId;
            }
            throw new FilmException("Primary key was not generated");
        } catch (Exception ex) {
            throw new FilmException(ex.getMessage());
        }
    }

    @Override
    public boolean update(Film film) {
        String sql = "update film set title = ?, released = ?, genre = ?,
                 stock = ?, version = version + 1 where id = ? and version = ?";
        try (Connection connection = DriverManager.getConnection(
                                                        url, username, password);
             PreparedStatement statement = connection.prepareStatement(sql)) {
            connection.setTransactionIsolation(
                                        Connection.TRANSACTION_READ_UNCOMMITTED);
            statement.setString(1, film.getTitle());
            //setDate takes a java.sql.Date argument
            statement.setDate(2, Date.valueOf(film.getReleased()));
            statement.setString(3, film.getGenre().toString());
            statement.setInt(4, film.getStock());
            statement.setLong(5, film.getId());
            statement.setInt(6, film.getVersion());
            return statement.executeUpdate() == 1;
        } catch (Exception ex) {
            throw new FilmException(ex.getMessage());
        }
    }
```

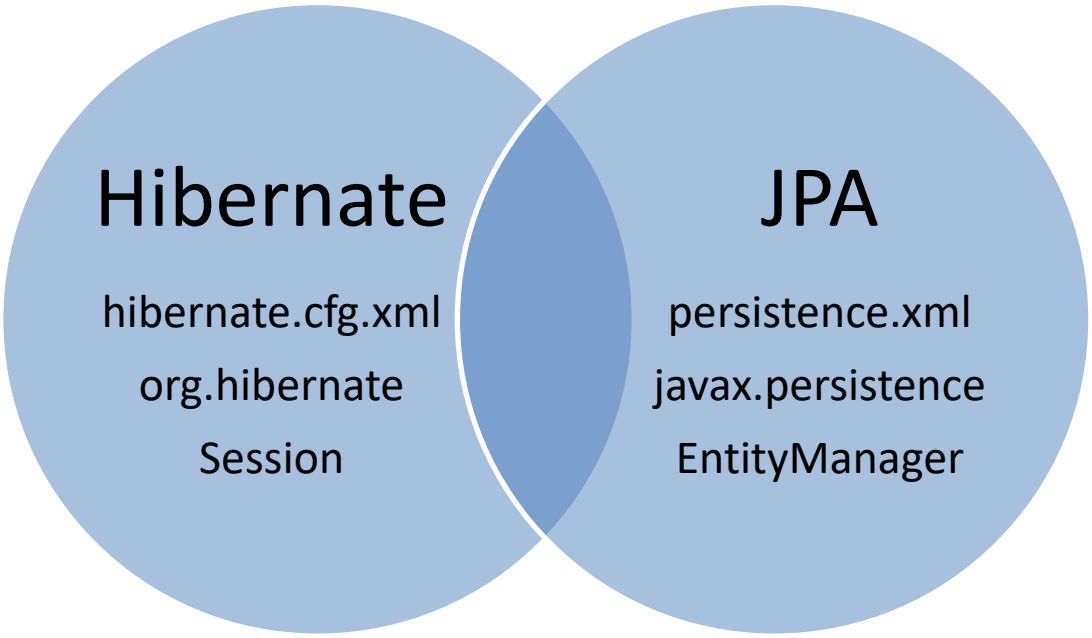
```
@Override
    public Collection<Film> selectByTitle(String search) {
        Collection<Film> films = new ArrayList<>();
        String sql = "select * from film where lcase (title) like ?";
        try (Connection connection = DriverManager.getConnection(
                                                url, username, password);
             PreparedStatement statement = connection.prepareStatement(sql)) {
            statement.setString(1, "%" + search.toLowerCase() + "%");
            ResultSet rs = statement.executeQuery();
            while (rs.next()) {
                Film film = toFilm(rs);
                films.add(film);
            }
        } catch (Exception ex) {
            throw new FilmException(ex.getMessage());
        }
        return films;
    }

    private Film toFilm(ResultSet rs) throws SQLException {
        Film film = new Film();
        film.setId(rs.getLong("id"));
        film.setStock(rs.getInt("stock"));
        //convert a java.sql.Date to a LocalDate
        film.setReleased(rs.getDate("released").toLocalDate());
        film.setTitle(rs.getString("title"));
        film.setGenre(Genre.valueOf(rs.getString("genre")));
        return film;
    }
}
```

Complete the delete, selectAll and selectById methods

```
@Override
public boolean delete(Long filmId) {
    String sql = "delete from film where id = ?";


@Override
public Collection<Film> selectAll() {
    Collection<Film> films = new ArrayList<>();
    String sql = "select * from film";


@Override
public Film selectById(Long id) {
    String sql = "select * from film where id = ?";
```

JPA

Hibernate is an implementation and an extension of the JPA specification.

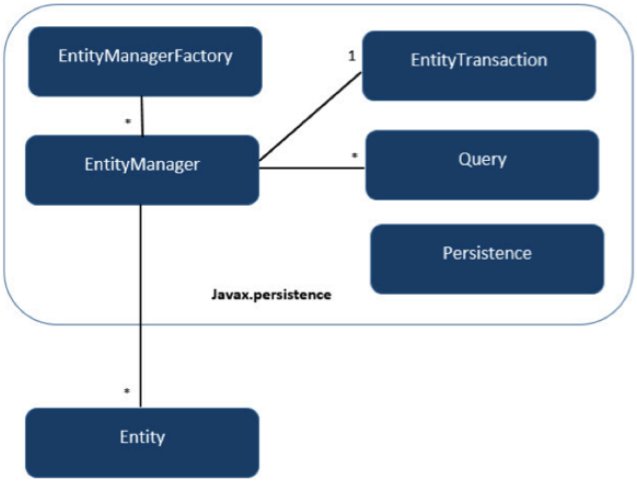


Object mapping

Using JDBC to interact with a database involves writing SQL expressions and translating between tabular data and objects. An object mapping framework presents relational data as Java objects, resulting in considerably less code.

The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

Include the hibernate-java8 dependency to enable mapping of java.time types




```
dependencies {
    compile 'org.hibernate:hibernate-entitymanager:5.2.5.Final'
    compile 'org.hibernate:hibernate-java8:5.2.5.Final'
    compile 'javax.transaction:jta:1.1'
```

In the project settings, add the ee7 api Javadoc for org.hibernate.javax.persistence

persistence.xml

The persistence.xml file is used to configure the EntityManager. It's placed in the src/main/java/resources/META-INF directory

```
<persistence version="2.1">
   <persistence-unit name="pu1" transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
      <class>entity.Film</class>
      <properties>
         <property name="javax.persistence.jdbc.driver"
                                          value="com.mysql.jdbc.Driver" />
         <property name="javax.persistence.jdbc.url"
                           value="jdbc:mysql://localhost:3306/filmstore" />
         <property name="javax.persistence.jdbc.user" value="root" />
         <property name="javax.persistence.jdbc.password" value="carpond" />

         <property name="hibernate.dialect"
                           value="org.hibernate.dialect.MySQLDialect" />
         <!-- Echo all executed SQL to stdout -->
         <property name="hibernate.show_sql" value="true" />
         <!-- Drop and re-create the database schema on startup -->
         <!--create-drop creates the tables when the SessionFactory is created.
            and drops them when the SessionFactory is closed explicitly. Other
properties
            are update, create, validate -->
         <property name="hibernate.hbm2ddl.auto" value="validate" />
      </properties>
   </persistence-unit>
```

Annotating the Film class

Hibernate takes a configuration-by-exception approach for annotations. For example, a class maps to a table with the same name. Some of the default mappings between Java and SQL types are shown below. Where there's ambiguity, an annotation is required. For example, an enumeration will map to a String when annotated with `@Enumerated(EnumType.STRING)`, or to an Integer when annotated with `@Enumerated(EnumType.ORDINAL)`

Java type	SQL Type
int	INTEGER
long	BIGINT
double	DOUBLE
boolean	BIT
String	VARCHAR

```
import javax.persistence.*;


@Entity //specifies that the class is an entity
public class Film {

	//@Id specifies the primary key of an entity
	@Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id;
	@Enumerated(EnumType.STRING) private Genre genre;
	@Version private int version;

	private LocalDate released;
	private int stock;
	private String title;
```

Genre enum members

To generate distinct comma delimited members for the Genre enum, use the following SQL expression:

```
select distinct concat(genre,',') from filmstore.film;
```

Singleton design pattern

The EntityManager is obtained from an EntityManagerFactory. An EntityManager instance is associated with a persistence context. A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed. The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities.

```
public class EntityManagerUtil {
   //calls private constructor when class is first loaded
   private final static EntityManagerUtil instance =  new EntityManagerUtil();
   //initialised by constructor
   private final EntityManagerFactory factory;

   //private constructor sets EntityManagerFactory
   private EntityManagerUtil() {
      factory = Persistence.createEntityManagerFactory("pu1");
   }

   //returns an EntityManager
   public static EntityManager getEntityManager() {
      return instance.factory.createEntityManager();
   }
}
```

CRUD operations

(create read update delete)

While unit tests focus on one unit of code, such as a class or method, integration tests involve multiple classes and layers of the application. The test class follows the convention of including “IT” in its name. It will run in the Maven integration-test phase, which can be started with `>gradle check`

Taking a TDD approach, the test method is written first and will initially fail.

```
   public class JpaFilmDAOIT {

       // arrange
       private FilmDAO dao = new JpaFilmDAO();

       @Test
       public void selectByIdShouldReturnCorrectFilmFromStore() {
          // act
          Film film = dao.selectById(5L);
          // assert
          assertEquals("Pulp Fiction", film.getTitle());
       }
```

Next, write the implementing method so that the assertion passes.

```
import entity.Film;

public class JpaFilmDAO implements FilmDAO {

       @Override
       public Film selectById(Long id) {
           EntityManager em = EntityManagerUtil.getEntityManager();
           // get returns null if id not in database
           Film film = em.find(Film.class, id);
           em.close();
           return film;
       }


        /* A JPQL Select Statement BNF select_statement ::= select_clause
         * from_clause [where_clause] [orderby_clause]  */
       @Override
       public Collection<Film> selectAll() {
              EntityManager em = EntityManagerUtil.getEntityManager();
              String jpql = "select f from Film f order by f.title";
              TypedQuery<Film> query = em.createQuery(jpql, Film.class);
              Collection<Film> films = query.getResultList();
              em.close();
              return films;
       }
```

```
/*  A JPQL Select Statement BNF select_statement ::= select_clause
 *   from_clause [where_clause] [orderby_clause] */
@Override
public Collection<Film> selectByTitle(String search) {
	EntityManager em = EntityManagerUtil.getEntityManager();
	String jpql = "select f from Film f where lower(f.title) like
	:searchText order by f.title";
	TypedQuery<Film> query = em.createQuery(jpql, Film.class);
	query.setParameter("searchText", "%" + search.toLowerCase() + "%");
	Collection<Film> films = query.getResultList();
	em.close();
	return films;
}
```

Objects, in relation to a session, can be transient, persistent, detached or removed. The save method persists a transient instance; an object that the database has no knowledge of. Changes to a persistent object are written to the database when the transaction commits.

State	Description
Transient	Not managed by hibernate. Call persist() to make the object persistent
Persistent	Changes to the object are written to the database when the transaction commits. Call find() to retrieve an entity from the database
Detached	Representation exists in database, but entity isn't persistent. Call merge() to make the entity persistent
Removed	Calling remove() will remove the database representation of the entity when the transaction commits

```
@Override
public Long insert(Film film) {
	Long id = 0L;
	EntityManager em = EntityManagerUtil.getEntityManager();
	em.getTransaction().begin();
	em.persist(film);// persists a transient instance
	id = film.getId();
	em.getTransaction().commit();// updates the database from the
	persistence context
	em.close();
	return id;
}
```

```
 /*The merge method changes an entity's state from detached or transient
 * to persistent. This will update a row with a matching primary key, or
 * insert a row if there's no match */

@Override
public boolean update(Film film) {
	EntityManager em = EntityManagerUtil.getEntityManager();
	em.getTransaction().begin();
	em.merge(film);// a detached entity is changed to persistent
	em.getTransaction().commit();
	em.close();
	return true;
}
```

```
	/**
	 * The delete method takes a persistent argument or a transient object with
	 * an id matching an id in the database. The row in the database is deleted
	 * when the transaction commits
	 */
	@Override
	public boolean delete(Long filmId) {
		EntityManager em = EntityManagerUtil.getEntityManager();
		EntityTransaction tx = em.getTransaction();
		tx.begin();
		try {
		   Film film =  em.find(Film.class, filmId);
		   if (film == null)
		      return false; // will execute finally next
		   em.remove(film);
		   em.getTransaction().commit();
		   return true;
		} catch (Exception e) {
		   if (tx != null)
		      tx.rollback();
		   throw e;
		} finally {
		   em.close();
		}
	}
```

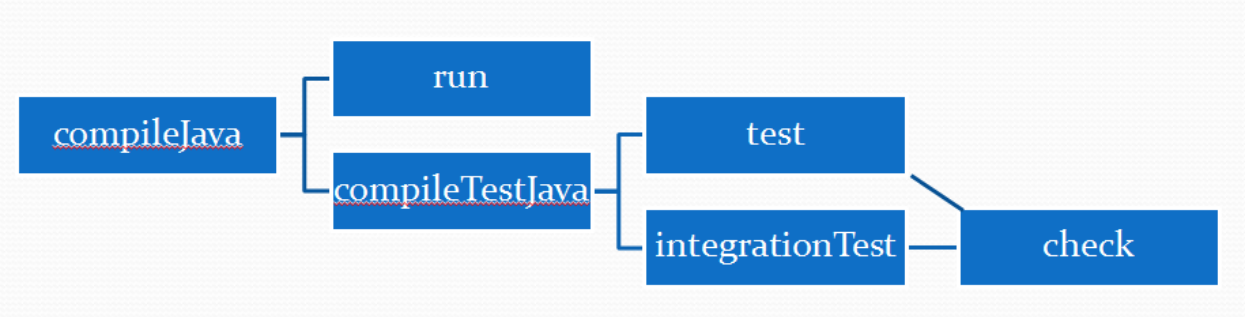
Integration test task

Gradle can be configured so that classes ending with "IT" are executed by a test task named integrationTest.

If includes are not provided, then all files will be included. Similarly, If excludes are not provided, then no files will be excluded.

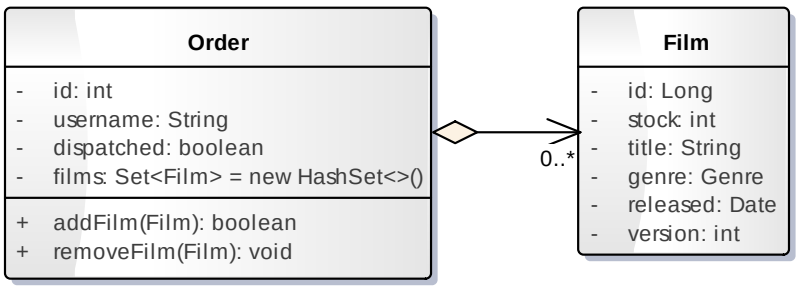
The check task depends on test and integrationTest tasks.

```
//configure the java plugin's test task
test {
    //exclude classes ending with IT from the test task
    exclude '**/*IT.class'
}

//add a test task named integrationTest that includes classes ending with IT
task integrationTest(type: Test){
    include '**/*IT.class'
}

check.dependsOn integrationTest
```

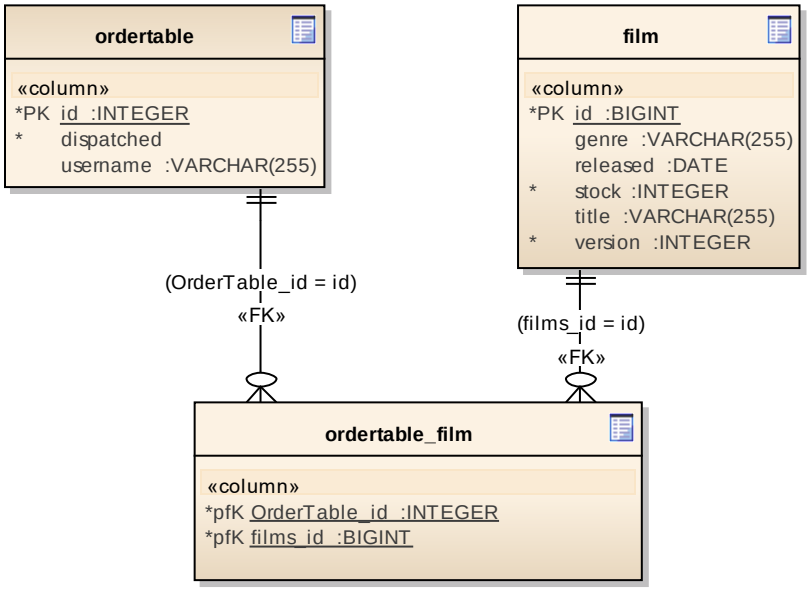

compileJava
run
compileTestJava
test
integrationTest
check

Associations

The class diagram shows an aggregation; a variant of the “has a” association relationship. Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong life cycle dependency on the container.



This association can be modelled in the database with a join table. A join table is typically used in the mapping of many-to-many and one-to-many associations.



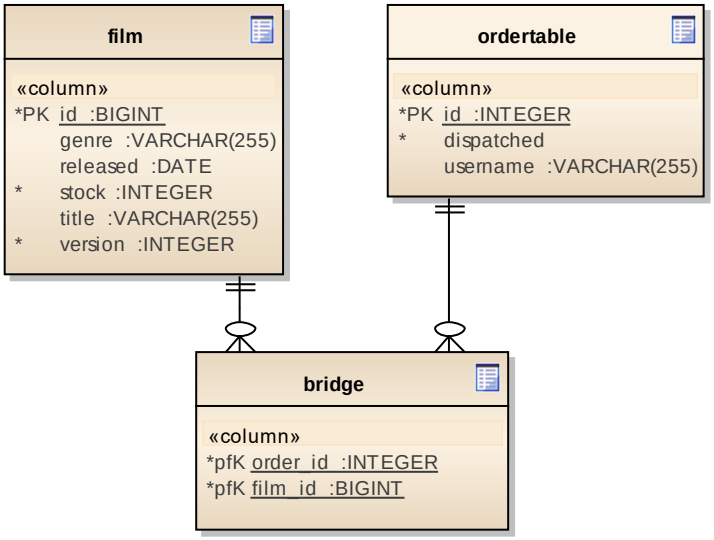

```
@Entity
@Table(name = "OrderTable")
public class Order {
	private boolean dispatched;

	@ManyToMany
	private Set<Film> films = new HashSet<>();

	@Id @GeneratedValue private int id;
	private String username;
```

The @JoinTable annotation is optional. If it's missing, the default values apply:

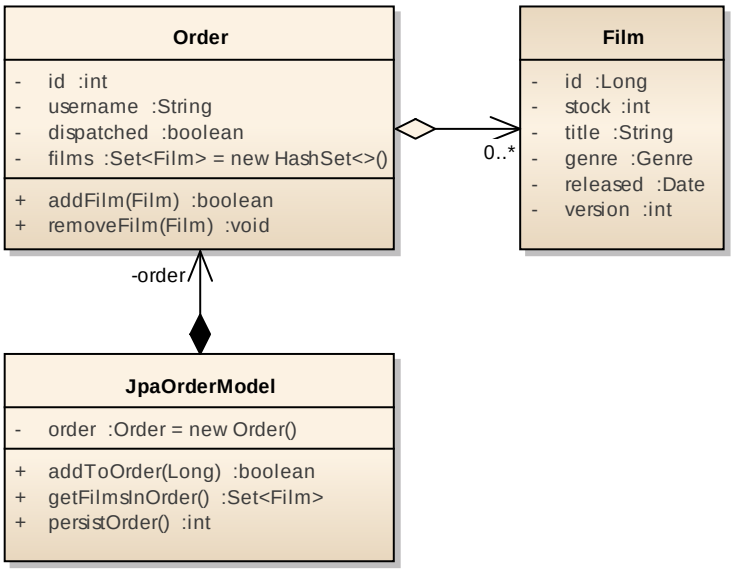
1. name is owningTable_associatedTable
2. joinColumns are the foreign key columns of the join table which reference the owning side of the association
3. inverseJoinColumns are the foreign key columns of the join table which reference the primary table of the entity that does not own the association

This would be written explicitly as follows:

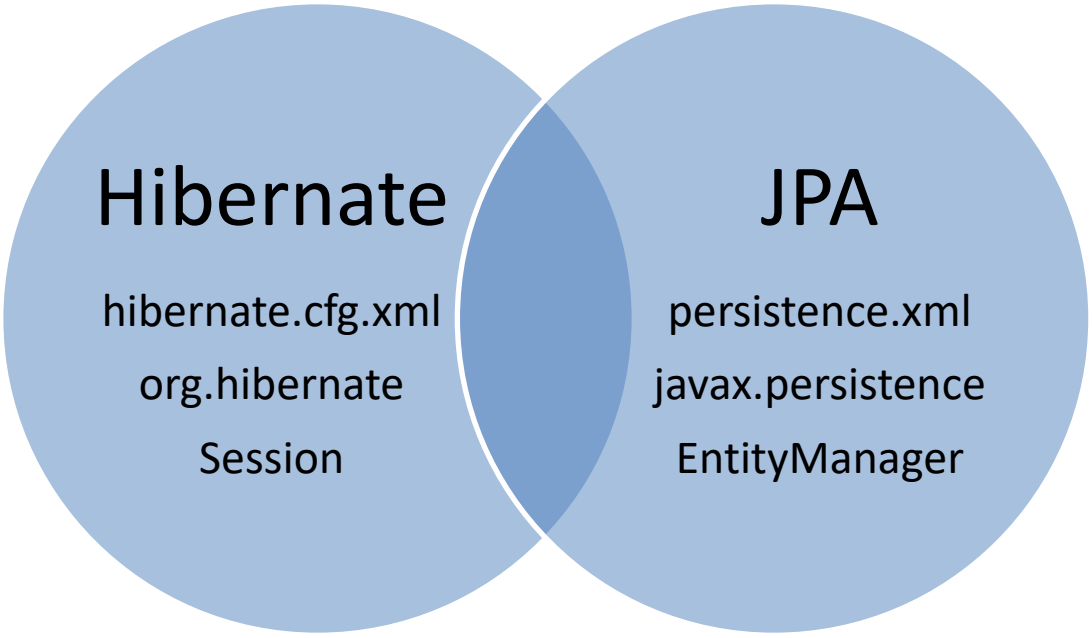
```
@JoinTable(name = "bridge",
           joinColumns = @JoinColumn(name = "order_id"),
           inverseJoinColumns = @JoinColumn(name = "film_id"))
@ManyToMany
private Set<Film> films = new HashSet<>();
```



OrderModel



A JpaOrderModel instance is intended to be associated with an individual user, enabling films to be added and removed from an Order and persisting the Order to the database. Hibernate is an implementation of the JPA specification; the class uses JPA to interact with the database.



1. Add the @Ignore annotation to JpaFilmDAOIT and update the persistence.xml file

```
<class>entity.Order</class>
<property name="hibernate.hbm2ddl.auto" value="create" />
```

2. Once the tables have been created, remove the @Ignore attribute and change the auto property back to validate; otherwise the tables will be recreated before each test and the Film table will be empty.

```
<property name="hibernate.hbm2ddl.auto" value="validate" />
```

3. Write some integration tests

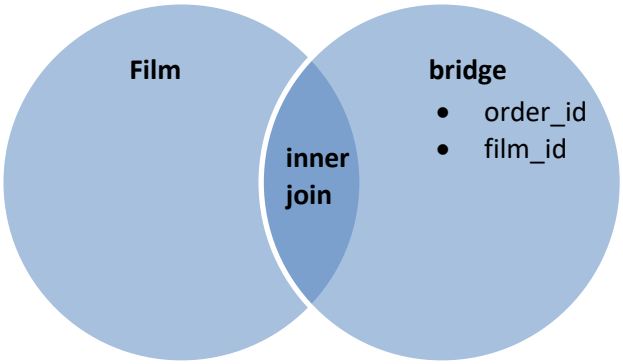
```
public class JpaOrderModelIT {
    // arrange
    private JpaOrderModel orderModel = new JpaOrderModel();

    @Test
    public void persistOrderShouldModifyDataStore() {
           //arrange
           orderModel.addToOrder(1L); //out of stock
           orderModel.addToOrder(2L); //stock 7
           orderModel.addToOrder(3L); //stock 8

           JpaFilmDAO hfd = new JpaFilmDAO();

           // act
           int id = orderModel.persistOrder();
           logger.info("order id "+id);
           Film film1 = hfd.selectById(1L);
           Film film2 = hfd.selectById(2L);
           Film film3 = hfd.selectById(3L);
           //dao.removeOrder(id);

           // assert
           assertEquals(2, orderModel.getFilmsInOrder().size());
           assertEquals(0, film1.getStock());
           assertEquals(6, film2.getStock());
           assertEquals(7, film3.getStock());
    }


    private static String[] commands = {
           "set foreign_key_checks = 0;",
           "create table if not exists Film ...
           "truncate table film;",
           "delete from film where id > 250;",
           "insert into Film ...
           "insert into Film ...
    };
```

4. Write a class that passes the tests

```
public class JpaOrderModel {

    private Order order = new Order();

    public boolean addToOrder(Long filmId) {
        EntityManager em = EntityManagerUtil.getEntityManager();
        try {
            Film film = em.find(Film.class, filmId);
            if (film == null || order.getFilms().contains(film))
                return false;
            order.addFilm(film);
            return true;
        } finally {
            em.close();
        }
    }

    public Set<Film> getFilmsInOrder() {
        return order.getFilms();
    }

    public int persistOrder() {
        EntityManager em = EntityManagerUtil.getEntityManager();
        em.getTransaction().begin();

        updateFilmStock(em);

        // makes the order object persistent, running a sql insert
        em.persist(order);
        order.getId();
        em.getTransaction().commit();
        em.close();
        return order.getId();
    }


    private void updateFilmStock(EntityManager em) {
        String ql = "update Film f set f.stock = f.stock - 1
                    where f.stock > 0 and f.id in :filmsInOrder";
        Query query = em.createQuery(ql);

        //lambda expression gets filmIds in customer's order
        List<Long> filmIds = order.getFilms().stream().
            map(f -> f.getId()).collect(Collectors.toList());
        query.setParameter("filmsInOrder", filmIds);
        int rowsUpdated = query.executeUpdate();

        //removeIf removes elements of a collection that satisfy the
        //predicate using an iterator, so avoiding a
        //ConcurrentModificationException
        order.getFilms().removeIf(f->f.getStock()==0);
    }
}
```

http://en.wikibooks.org/wiki/Java_Persistence/JPQL_BNF

Table joins



Because SQL is based on set theory, each table can be represented as a circle in a Venn diagram. The ON clause in the SQL SELECT statement that specifies join conditions determines the point of overlap for those circles and represents the set of rows that match. For example, in an inner join, the overlap occurs within the interior or "inner" portion of the two circles. An outer join includes not only those matched rows found in the inner cross section of the tables, but also the rows in the outer part of the circle to the left or right of the intersection.

The following expression retrieves the film titles within an order with an id of 1.

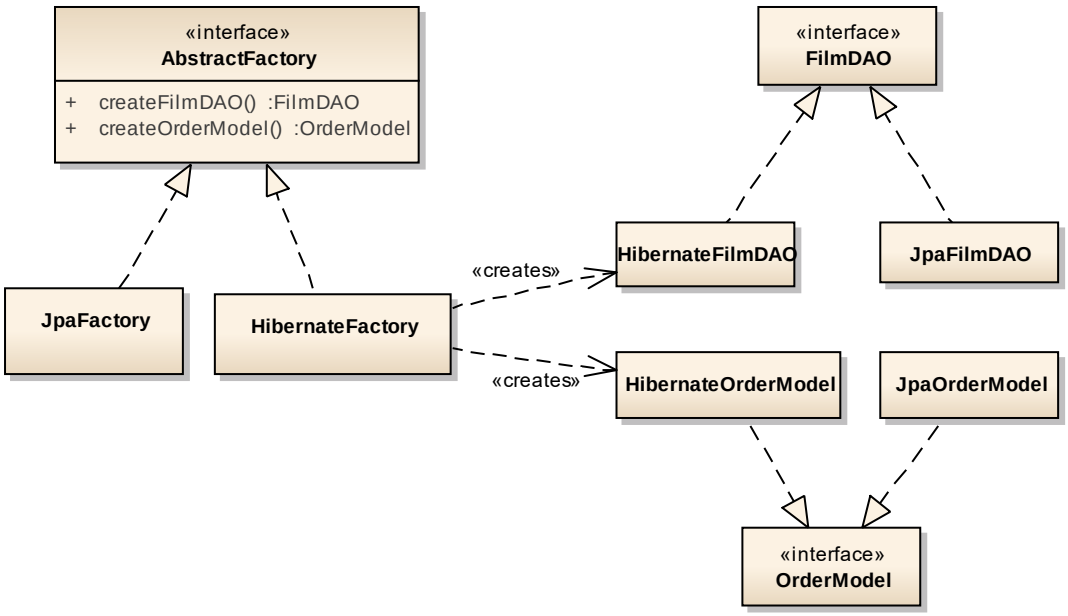
```
select title from Film
inner join bridge on Film.Id = bridge.film_id
where bridge.order_id = 1;
```

Navigating between related entities

- select o.films from Order o where o.id = :id
- select o from Order o, in (o.films) as f where f.id = :id

Abstract Factory

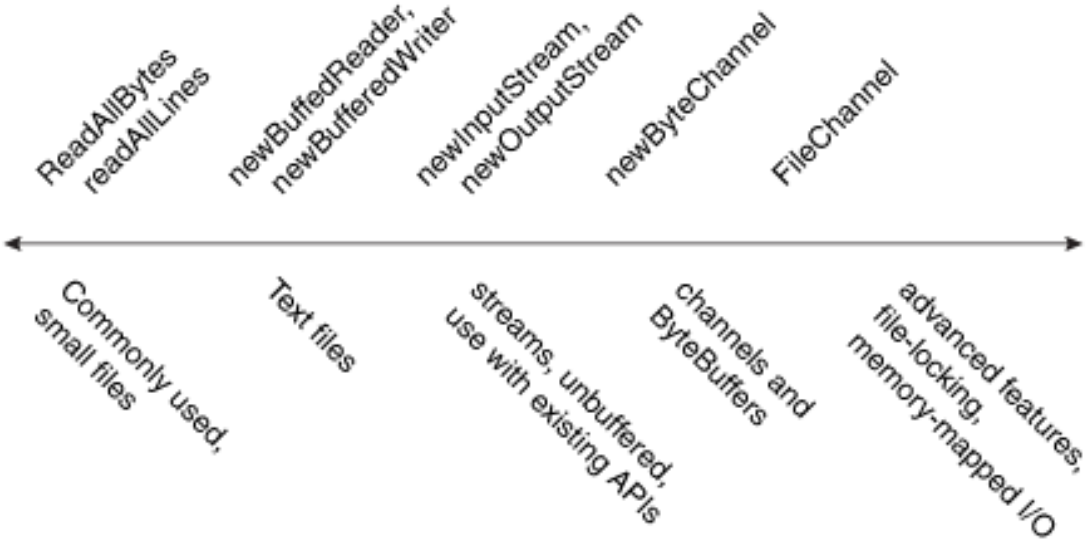
The abstract factory pattern provides an interface for creating families of related objects without specifying their concrete classes



Streams

Reading, Writing, and Creating Files

This diagram arranges the file I/O methods by complexity



Small Files

```
Path path = Paths.get("file.txt");
Set<String> zoneIds = ZoneId.getAvailableZoneIds();
Files.write(path, zoneIds, StandardOpenOption.CREATE);

List<String> lines = Files.readAllLines(path);
lines.forEach(System.out::println);
```

Text Files

```
Path path = Paths.get("file.txt");
Charset charset = Charset.forName("UTF-8");
String s = "something";
try (BufferedWriter writer = Files.newBufferedWriter(path, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}

try (BufferedReader reader = Files.newBufferedReader(path, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Unbuffered Streams

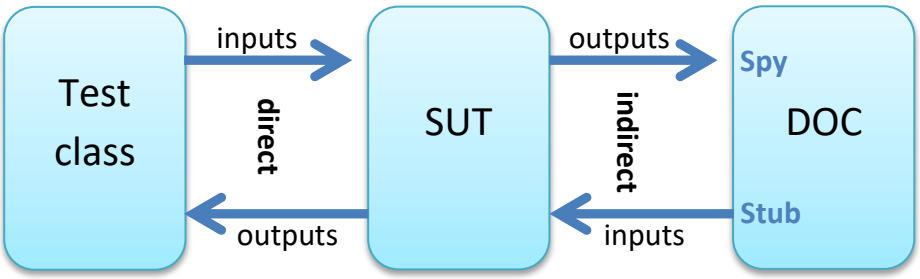
```
public class Main {
   public static void main(String[] args) {
      Film film = new Film("There's something about Mary",
            2, LocalDate.of(1997, 1, 27), Genre.COMEDY);
      Path path = Paths.get("object.bin");

      //Serialize object
      try (ObjectOutputStream oos = new ObjectOutputStream(
            Files.newOutputStream(path))) {
         oos.writeObject(film);
      } catch (IOException e) {
         System.out.println(e);
      }

      //Deserialize object
      if (!Files.exists(path))
         return;
      try (ObjectInputStream ois = new ObjectInputStream(
            Files.newInputStream(path))) {
         Film f = (Film) ois.readObject();
         System.out.println(f);
      } catch (Exception e) {
         System.out.println(e);
      }
```

Serializable interface

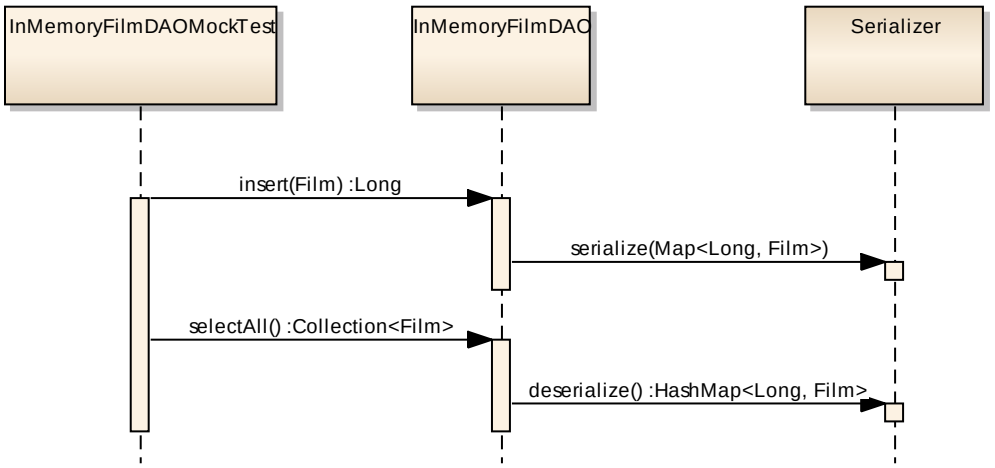
Serialized objects must implement the java.io.Serializable interface. To ensure matching versions of classes, including a serialVersionUID field in the serialized class is recommended. An InvalidClassException is thrown if the serial version of the class does not match that of the class descriptor read from the stream.

```
public class Film implements Serializable {
      private static final long serialVersionUID = 1L;
```

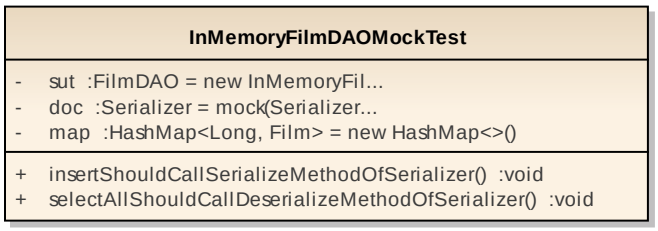
Interactions testing



Unit tests apply to classes in isolation; the System Under Test (SUT) in the above diagram. They are intended to run fast and to pinpoint bugs with accuracy. **State testing** involves writing tests for direct inputs and outputs, while **interactions testing** verifies the way that the SUT interacts with collaborators (Depended On Components or DOCs).



The above sequence diagram illustrates the test class calling the insert and selectAll methods of the SUT, while the SUT interacts with the Serializer class.



Mockito

Mockito is an open source testing framework for Java, enabling the creation of test double objects in automated unit tests for the purpose of Test-driven Development.

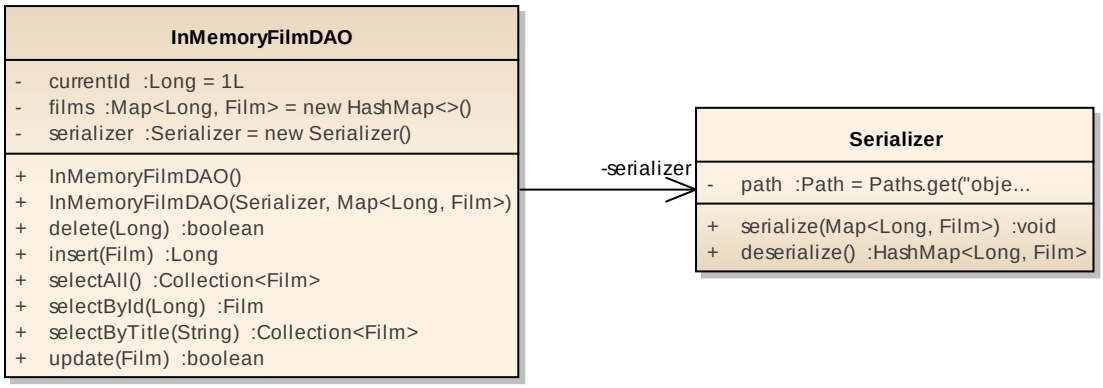
Add the gradle dependency for Mockito

```
    testCompile 'org.mockito:mockito-core:2.5.0'
```

```
package session;

public class InMemoryFilmDAOTest {

   // arrange
   private Map<Long, Film> films = new HashMap<>();
   private FilmDAO sut = new InMemoryFilmDAO(films);

      @Test
   public void insertShouldCallSerializeMethodOfSerializer() {
      //arrange
      Serializer doc = mock(Serializer.class);
      FilmDAO sut = new InMemoryFilmDAO(films, doc);
      // act
      sut.insert(new Film());
      // assert
      verify(doc).serialize(films);  //doc is a spy (verifies indirect outputs)
   }

   @Test
   public void selectAllShouldCallDeserializeMethodOfSerializer() {
      //arrange
      Serializer doc = mock(Serializer.class);
      FilmDAO sut = new InMemoryFilmDAO(films, doc);
      Film film1 = new Film("The Pink Panther", 5,
                                 LocalDate.of(1964, 1, 20), Genre.COMEDY);
      films.put(1L,film1);
      when(doc.deserialize()).thenReturn(films);

      // act
      Collection<Film> filmsCollection = sut.selectAll();

      // assert
      verify(doc).deserialize(); //doc is a stub (verifies indirect inputs)
      assertEquals(1, filmsCollection.size());
   }
}
```

Serialization

Serialization is the process of translating object state into a format that can be stored, for example in a file, or transmitted across a network.




```
public class SerializerImpl implements Serializer {
   private Path path = Paths.get("object.bin");

   //Serialized objects must implement the java.io.Serializable interface
   @Override
   public void serialize(Map<Long, Film> films) {
      try (ObjectOutputStream oos = new ObjectOutputStream(
            Files.newOutputStream(path))) {
         oos.writeObject(films);
      } catch (IOException e) {
         System.out.println(e);
         throw new FilmException(e.getMessage());
      }
   }

   @Override
   public Map<Long, Film> deserialize() {
      if (!Files.exists(path))
         return new ConcurrentHashMap<Long, Film>();
      try (ObjectInputStream ois = new ObjectInputStream(
            Files.newInputStream(path))) {
         return (Map<Long, Film>) ois.readObject();
      } catch (Exception e) {
         System.out.println(e);
         throw new FilmException(e.getMessage());
      }
   }
}
```

Generating a sequential id

```
OptionalLong optional = films.values().stream().mapToLong(f -> f.getId()).max();
id = optional.isPresent() ? optional.getAsLong() + 1 : 1;
```

OptionalLong is a container object which may or may not contain a long value. If a value is present, isPresent() will return true and getAsLong() will return the value.

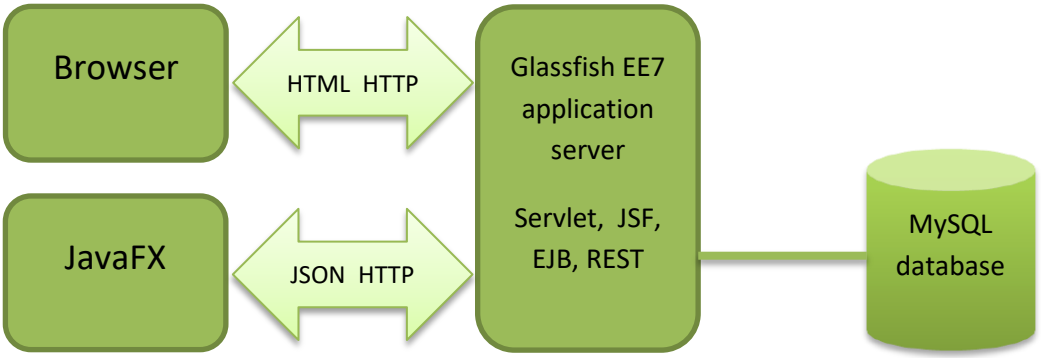
Synchronized block

```
public class InMemoryFilmDAO implements FilmDAO {

    @Override
    public Long insert(Film film) {
        Long id = null;
        synchronized (this) {
            OptionalLong optional = films.values().stream().
                                            mapToLong(f -> f.getId()).max();
            id = optional.isPresent() ? optional.getAsLong() + 1 : 1;
            film.setId(id);
            films.putIfAbsent(id, film);
        }
        if (serializer != null)
            serializer.serialize(films);
        return id;
    }
```

Web applications

http://gradle.org/docs/current/userguide/war_plugin.html

Overview



GlassFish

Glassfish 4.1.1 seems to have some unresolved bugs with JAX-RS. Payara Server is intended to be a drop in replacement for GlassFish Server Open Source Edition, and is released quarterly with bug fixes.

1. start server
   payara41\bin
   asadmin start-domain

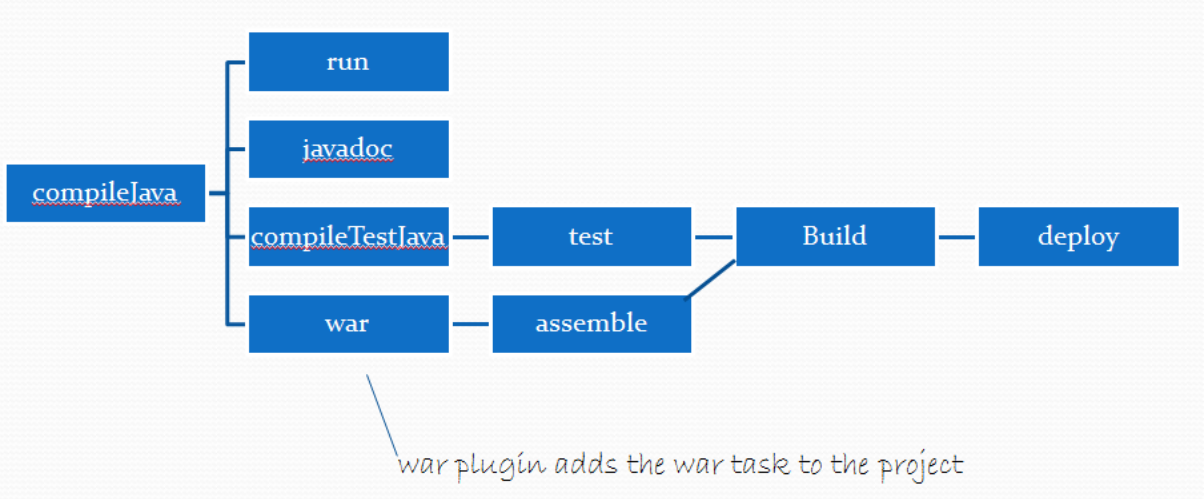
2. Logging
   olv.bat
   tail server.log at domains/domain1/logs/server.log

3. Open admin console
   http://localhost:4848

4. Deploy application by copying war to domains/domain1/autodeploy directory

5. Add the gradle war plugin to the build.gradle. The war task copies generates a web archive in build/libs. It depends on the compile task.

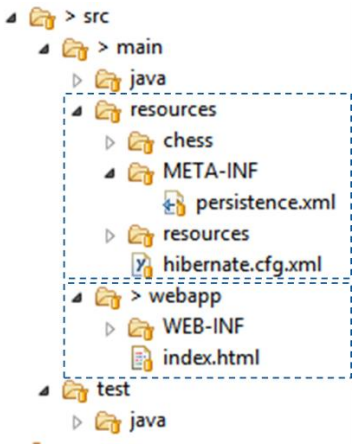
```
apply plugin:'application'
apply plugin:'war'
```



6. Add the Java EE 7 dependency to build.gradle. The War plugin adds two dependency configurations named providedCompile and providedRuntime. Those two configurations have the same scope as the respective compile and runtime configurations, except that they are not added to the WAR archive.

```
dependencies {
      providedCompile 'javax:javaee-api:7.0'
```

7. Add a source folder src/main/webapp



a. http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

b. Add index.html containing a form with the action Servlet1

```
<html>
    <head>
            <title>Home page</title>
    </head>
    <body>
            <form action="Servlet1" method="get">
                    <input type="text" name="searchText" />
                    <input type="submit" value="Servlet" />
            </form>
    </body>
</html>
```

8. Change hibernate hbm2ddl.auto property to validate, so tables aren't rebuilt
9. Add a task to copy the war file from the build directory to the glassfish autodeploy directory. This task will be executed after the build task

```
task deploy (type: Copy) {
    from 'build/libs'
    into '…domains/domain1/autodeploy'
    include '**/*.war'
}
deploy.dependsOn build
```

10. >gradle deploy generates the war file in the build/libs folder, asnd copies it to the glassfish autodeploy directory.
11. Launch application from admin console http://localhost:4848

Server log

1. view server log
    a. glassfish4\glassfish\domains\domain1\logs\server.txt
2. OtrosLogViewer
    a. Paste the glassfish.pattern into the plugins\logimporters folder
    b. click olv.bat to start
    c. select "tail glassfish"
    d. file:///C:/Users/user/Downloads/java_ee_sdk-7u1/glassfish4/glassfish/domains/domain1/logs/server.txt

Servlets

A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients over HTTP.

A subclass of HttpServlet overrides one or more of the following methods

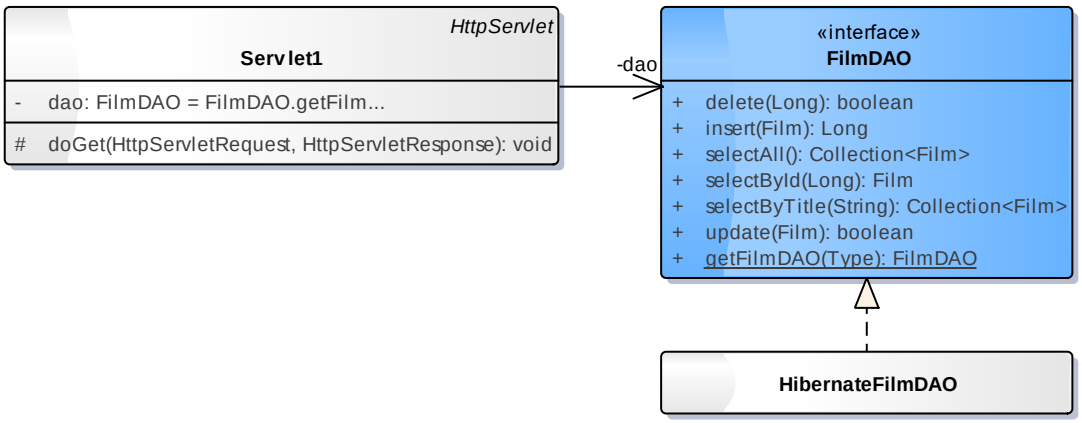
- doGet, for HTTP GET requests
- doPost, for HTTP POST requests
- doPut, for HTTP PUT requests
- doDelete, for HTTP DELETE requests

1. Add an HTML form
2. Add a class named Servlet1 to a package called servlets
    a. Extend HttpServlet
    b. Override doGet
    c. response.getWriter().print("Servlet1");
    d. Annotate class with @WebServlet("/Servlet1")

```
@WebServlet("/Servlet1")
public class Servlet1 extends HttpServlet{
	@Override
	protected void doGet(HttpServletRequest request,
				HttpServletResponse response)
					throws ServletException, IOException {
		response.getWriter().print("Servlet1");
	}
}
```

3. Generate a list of films from the servlet

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Hello World!</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Acme Film Store</h1>");
String str = req.getParameter("searchText");
Collection<Film> films = str == null ?
                                    dao.selectAll() :
                                    dao.selectByTitle(str);
for (Film film : films) {
	out.print(film.getTitle());
	out.println("<br/>");
}
out.println("</body>");
out.println("</html>");
```

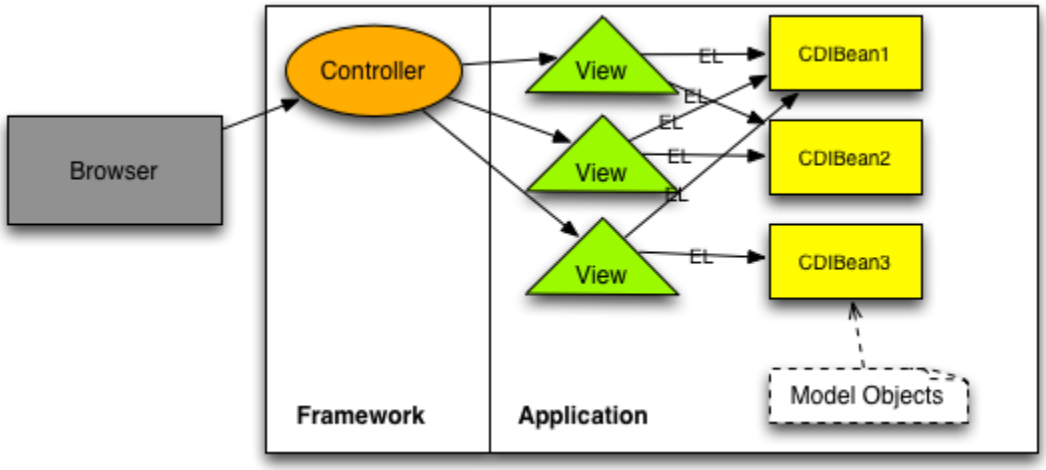


JSF

Overview

JavaServer Faces is a server-side component framework for building Java based web applications. It comprises

- An API for representing components and managing their state; handling events, server-side validation, and data conversion; page navigation and supporting internationalization
- Tag libraries for adding components to web pages and for connecting components to server-side objects



JSF is a MVC framework. CDI beans comprise the model; facelets comprise the view and the controller is FacesServlet, which manages the request processing lifecycle.

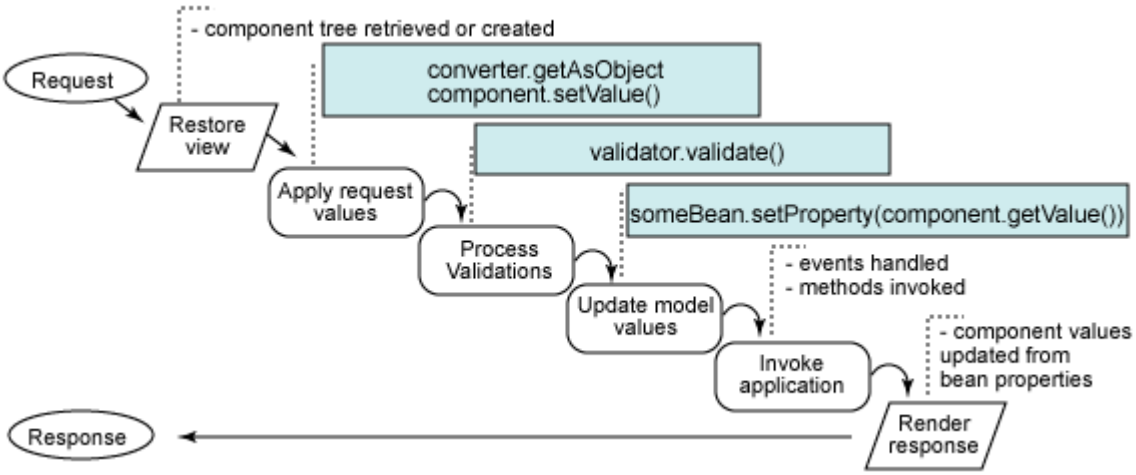
1. Add an page named filmlist.xhtml
    a. New facelet composition page
    b. Add a web-inf folder to src/main/webapp
    c. Add an empty file, faces-config.xml, to this folder
    d. Open http://localhost:8080/[war filename]/**facelet.jsf**

```
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
        xmlns:h="http://xmlns.jcp.org/jsf/html"
        xmlns:f="http://xmlns.jcp.org/jsf/core"
        xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<body>
    <h:form>
          <h:inputText />
          <h:commandButton value="Update" />
    </h:form>
    <ui:debug />
</body>
</html>
```

2. View component tree
    a. The ui:debug tag enables a popup displaying information about the JSF component tree and scoped variables in the browser. CTRL + SHIFT + D is the default value. The component classes are in javax.faces.component.html
    b. JSF API https://javaserverfaces.java.net/nonav/docs/2.2/javadocs/index.html

Lifecycle



- **Restore View** - For an initial request, an empty view is created. The life cycle then advances to the render response phase. If the request is a postback, the view is restored using the state information saved on the client or the server.
- **Apply Request Values** - Components extract their value from the request parameters. If necessary, the value is converted to the type of the associated backing bean property
- **Process Validations** - If validation fails, an error message is added to the FacesContext, and the life cycle advances directly to the render response phase
- **Update Model Values** - Backing bean properties are set
- **Invoke Application** - Event handling methods are executed
- **Render Response** - The components update their values from the backing bean and render themselves. The state of the response is saved so that it is available in the restore view phase for subsequent requests.

CDI

Backing beans enable the separation of application logic from presentation. Contexts and Dependency Injection beans are managed by the application server and are bound to a context, such as the current request or a browser session. CDI specifies mechanisms for injecting dependencies and handling events. Use the @Named annotation to declare a CDI bean. The argument is the variable used to reference the bean instance from a facelet. A deployment descriptor, beans.xml, is required in the WEB-INF folder; the file may be empty.

The scope of the bean can be

- RequestScoped – a new instance is created for each HTTP request
- SessionScoped – uses cookies or URL rewriting to maintain the session
- ApplicationScoped – shared among all requests and sessions

Note that these annotations are in the `javax.enterprise.context package.` Session scoped beans must be capable of being passivated, so should implement the Serializable interface.

Methods annotated with @PostConstruct are executed after the bean has been created and resources injected, while @PreDestroy methods are executed before the bean goes out of scope.

```
@Named("listBean")
@javax.enterprise.context.SessionScoped
public class ListBean implements Serializable {
	private FilmDAO dao = New JpaFilmDAO();
	private Collection<Film> films;
	private String searchText;

	@PostConstruct
	private void init(){
		films = dao.selectAll();
	}
	public Collection<Film> getFilms() {
		return films;
	}
	public String getSearchText() {
		return searchText;
	}
	public void setSearchText(String searchText) {
		this.searchText = searchText;
		films = dao.selectByTitle(searchText);
	}
}
```

Expression Language

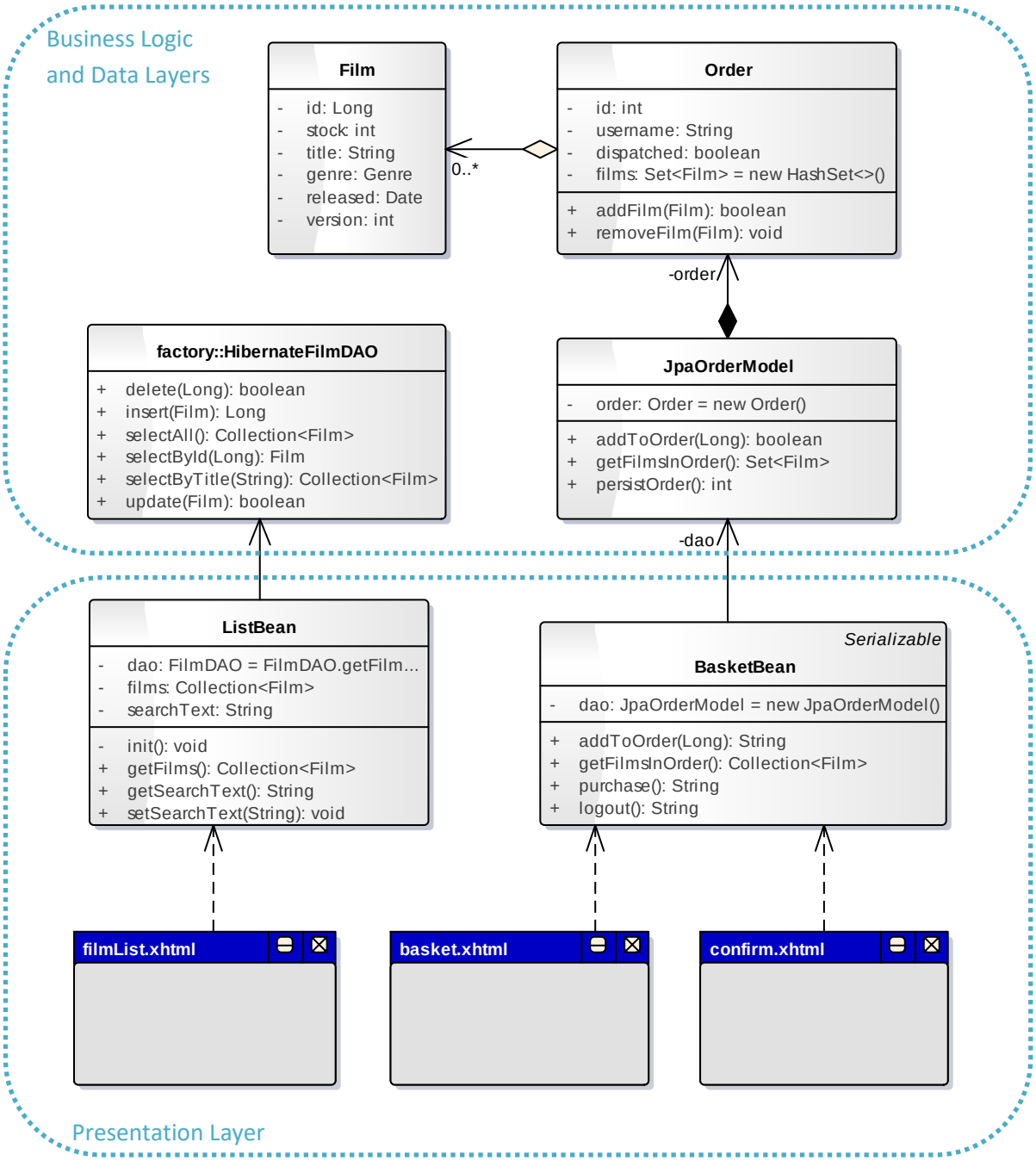
JSF Expression language enables facelets to access properties and methods of backing beans. Whether a get or set method is called depends on the type of the JSF component; an inputText component will call a set method, while an outputText component will call a get method

- #{listBean.searchText} calls the setSearchText method of the ListBean instance
- #{listBean.films} calls the getFilms method of the ListBean instance
- #{film.title} calls the getTitle method of the Film instance

filmList.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<body>
   <h:form>
       <h:inputText value="#{listBean.searchText}" />
       <h:commandButton value="Update" />
       <h:dataTable id="filmTable" value="#{listBean.films}" var="film">
          <h:column>
              <h:outputText value="#{film.title}" />
          </h:column>
       </h:dataTable>
   </h:form>
   <ui:debug />
</body>
</html>
```

Basket Bean



The JSF beans contain the application logic; these connect to the business logic and data layers.

BasketBean.java

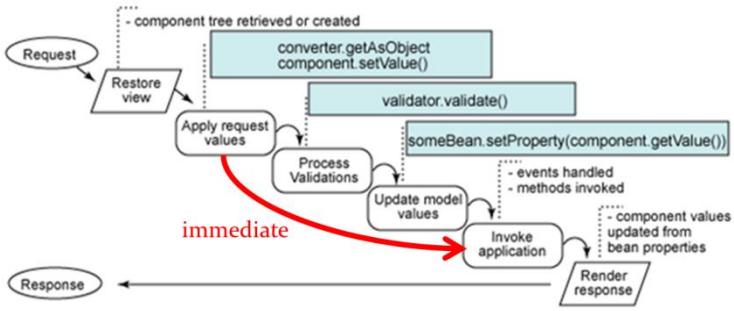
```
@Named
@javax.enterprise.context.SessionScoped
public class BasketBean implements Serializable {
	private JpaOrderModel dao = new JpaOrderModel();
	public String addToOrder(Long filmId) {
		boolean added = dao.addToOrder(filmId);
		return "basket.xhtml?faces-redirect=true";
	}
	public Collection<Film> getFilmsInOrder() {
		return dao.getFilmsInOrder();
	}
	public String purchase() {
		dao.persistOrder();
		return "confirm.xhtml";
	}
	public String logout() {
		((HttpSession) FacesContext.getCurrentInstance().
		getExternalContext().getSession(false)).invalidate();
		return "filmlist.xhtml?faces-redirect=true";
	}
}
```

Facelets

Change the outputText component in the filmlist.xhtml to a commandLink that forwards to the BasketBean backing bean.

```
<h:commandLink action="#{basketBean.addToOrder(film.id)}"
               immediate="true" value="#{film.title}" />
```

The immediate attribute causes events to be fired after the *apply request values* phase, bypassing the *update model values* phase.



basket.xhtml

```
<body>
  <h:form>
    <h:commandButton value="Film List" action="filmlist.xhtml" />
    <h:commandButton value="Purchase"
                     action="#{basketBean.purchase}" />
    <h:dataTable value="#{basketBean.filmsInOrder}" var="film">
      <h:column>
        #{film.title}
      </h:column>
    </h:dataTable>
  </h:form>
</body>
```

confirm.xhtml

```
    <body>
       <h1>Purchase confirmed</h1>
       <h:form>
          <h:dataTable value="#{basketBean.filmsInOrder}" var="film">
             <h:column>
                    #{film.title}
             </h:column>
          </h:dataTable>
          <h:commandButton action="#{basketBean.logout}" value="Logout" />
       </h:form>
    </body>
```

Localisation

1. Add resources.labels to src/main/resources
   Properties files contain key=value pairs. The file name ends with an underscore followed by the IANA two letter country code, for example labels_de.properties
2. Add loadBundle to facelet. This loads the resource bundle for the Locale of the current view and stores it as a Map in request scope.
3. Simulate german user by adding <f:view locale="en" /> to facelet

```
<f:loadBundle basename="resources.labels" var="labels" />
<f:view locale="de" />
<body>
	<h:form>
		<h:inputText value="#{listBean.searchText}" />
		<h:commandButton value="#{labels.updateButton}" />
```

Enterprise beans

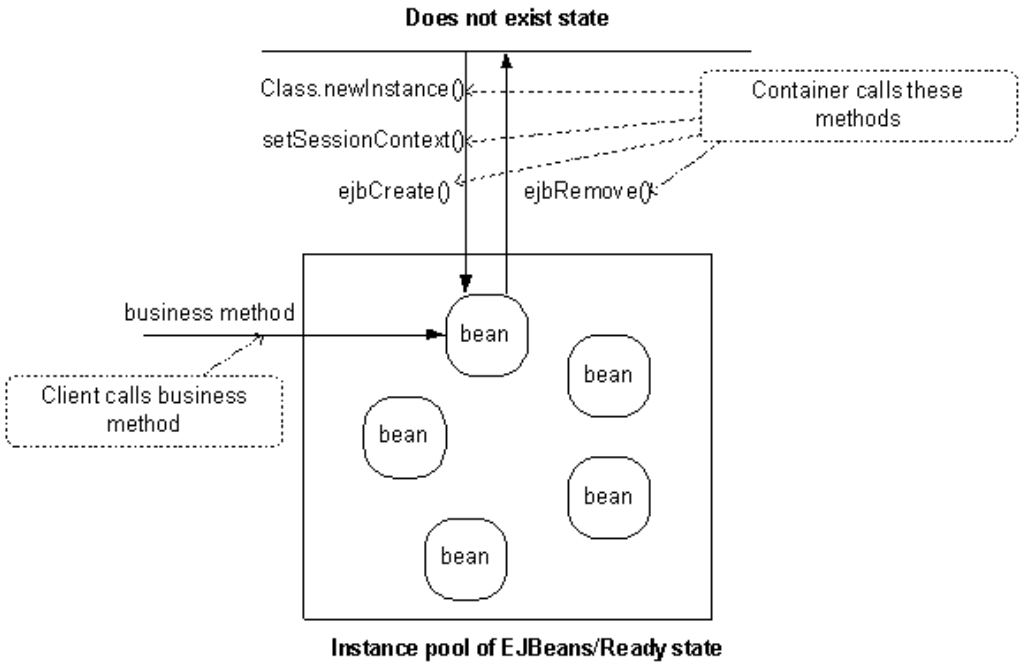
Enterprise JavaBeans are managed, server-side components for the modular construction of applications that are hosted by an application server. They encapsulate the business logic of an application, handling common concerns such as persistence, transactional integrity, and security

EE7 Web profile is a subset of EE7 that includes EJB Lite. The EJB Lite specification excludes EJBs with remote interfaces and Message Driven Beans.

Web Profile vs. Full Platform distributions
https://glassfish.java.net/webprofileORfullplatform31x.html

Stateless session beans

The application server maintains a pool of bean instances. When a client invokes methods of a bean, an instance is retrieved from the pool, and returned to the pool when the method is finished. All instances of are equivalent, and state is not preserved across different method calls. Since access to a bean instance is limited to one client at a time, they're automatically thread-safe. Because they can support multiple clients, stateless session beans can offer better scalability for applications that require large numbers of clients.



Stateless session bean life cycle

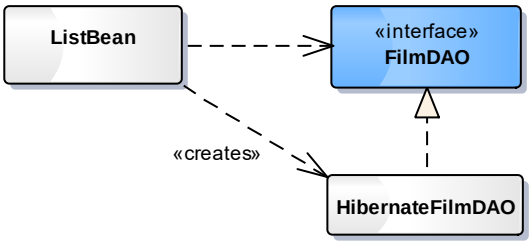
Annotate the class with Stateless. A qualifier is required as FilmDAO has more than one implementation

```
@Stateless
@Hibernate
public class JpaFilmDAO implements FilmDAO {

@Qualifier
@Target({TYPE, FIELD})
@Retention(RUNTIME)
public @interface Hibernate {
}
```

Use CDI dependency injection to get a reference to the Session bean from the JSF backing bean

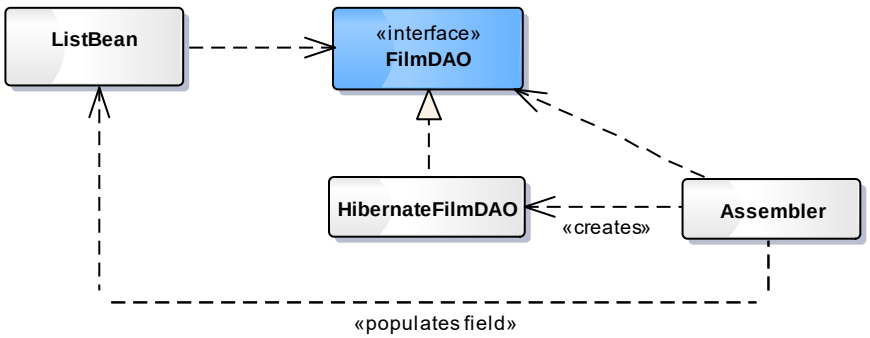
```
@Named("listBean")
@javax.enterprise.context.SessionScoped
public class ListBean  implements Serializable{
      //private FilmDAO dao = New JpaFilmDAO();
      @Inject @Hibernate private FilmDAO dao;
```

Inversion of control



By instantiating JpaFilmDAO, ListBean is dependent on an implementation rather than an abstraction.

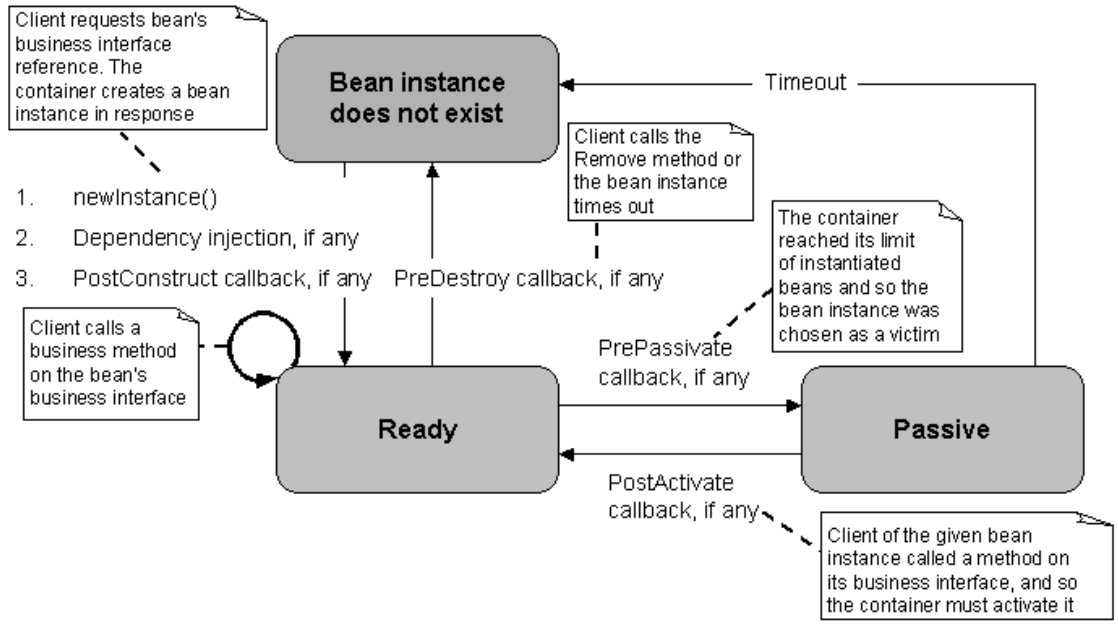
Dependency injection inverts the dependency of ListBean on the implementation of FilmDAO. This follows the design principle "**Depend upon abstractions. Do not depend upon concrete classes**"



See http://martinfowler.com/articles/injection.html

Stateful session beans

Instances aren't shared; state is maintained for a particular client for the duration of the session.



Annotate the class with Stateful

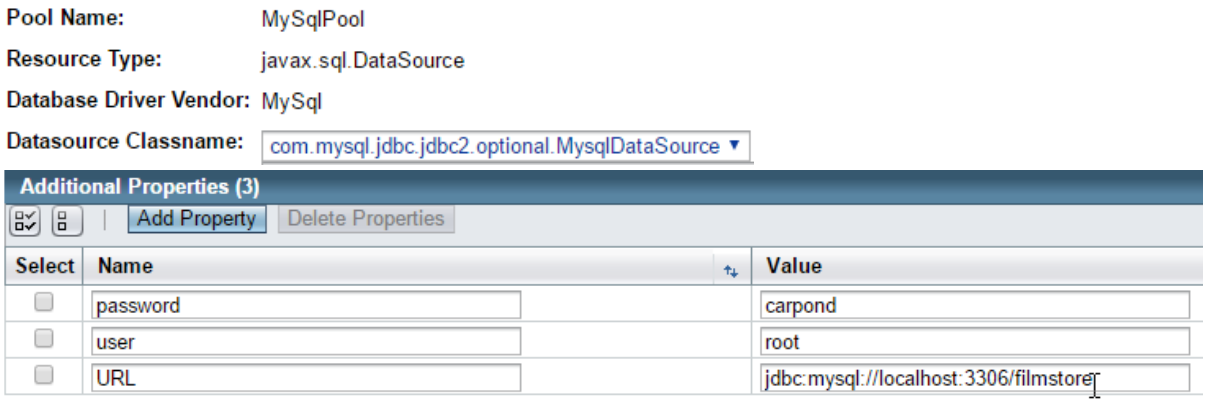
```
@Stateful
public class JpaOrderModel implements OrderModel {
```

Use CDI dependency injection to get a reference to the Session bean from the JSF backing bean

```
@Named
@javax.enterprise.context.SessionScoped
public class BasketBean implements Serializable {
	//private JpaOrderModel dao = new JpaOrderModel();
	@Inject private OrderModel dao;
```

JTA Persistence Context

1. Copy mysql-connector-java to server's glassfish\lib folder
2. Add a MySql JDBC connection pool



3. Associate a JNDI name (jdbc/filmstore) with the connection pool
4. Add a persistence unit to persistence.xml

```
<persistence-unit name="pu2" transaction-type="JTA">
   <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
   <jta-data-source>jdbc/filmstore</jta-data-source>
   <properties>
      <property name="hibernate.transaction.jta.platform"
     value="org.hibernate.service.jta.platform.internal.SunOneJtaPlatform"/>
      <property name="hibernate.dialect"
     value="org.hibernate.dialect.MySQLDialect" />
      <property name="hibernate.hbm2ddl.auto" value="validate"/>
   </properties>
</persistence-unit>
```

5. Inject the EntityManager and remove any code that closes the entity manager. The following is an additional implementation of the FilmDAO interface

```
@Stateless
@Jta
public class JtaFilmDAO implements FilmDAO {

    @PersistenceContext(unitName = "pu2") private EntityManager em;

    @Override
    public Film selectById(Long id) {
        Film film = em.find(Film.class, id);
        return film;
    }
```

6. Inject the EJBContext to enable rolling back a transaction

```
    @Resource private EJBContext context;
    @Override
    public boolean delete(Long filmId) {
        try {
            Film film =  em.find(Film.class, filmId);
            if (film == null)
                return false; // will execute finally next
            em.remove(film);
            return true;
        } catch (Exception e) {
            context.setRollbackOnly();
            return false;
        }
```

```
        }

    }
```

Web application security

1. Create users and groups in database
2. Set up a database realm
3. Map groups to roles in glassfish-web.xml
4. Configure web.xml
    a. Specify the web resources to be protected
    b. Define roles
    c. Specify authentication method
5. Provide login and error pages

Create users and groups in database

```
use filmstore;
create table USERTABLE( USERNAME varchar(20) primary key, PASSWORD varchar(256));

create table GROUPTABLE(USERNAME varchar(20) primary key, GROUPNAME varchar(20));

insert into GROUPTABLE(USERNAME,GROUPNAME) values ('user1', 'customer');
insert into GROUPTABLE(USERNAME,GROUPNAME) values ('user2', 'admin');

insert into USERTABLE(USERNAME,PASSWORD) values ('user1',
'5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8');

insert into USERTABLE(USERNAME,PASSWORD) values ('user2',
'5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8');
```

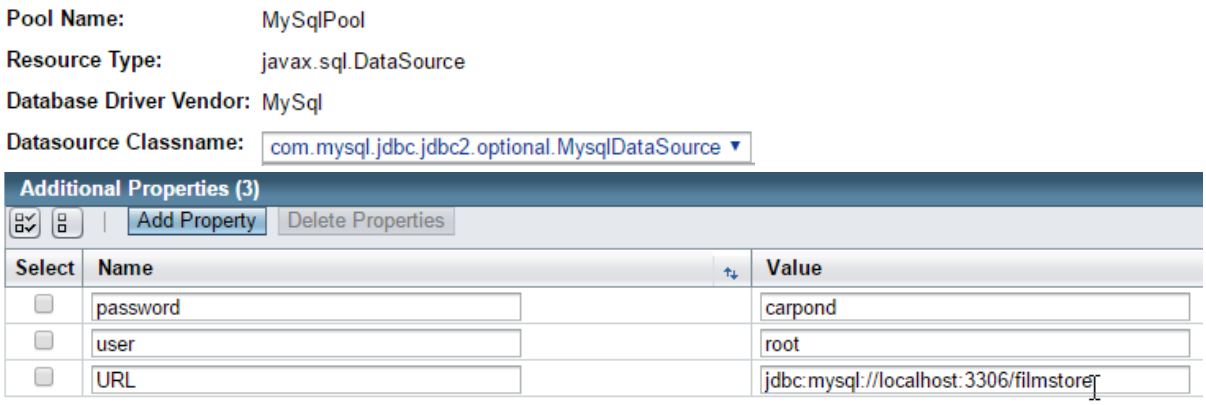
SHA-256 (Secure Hash Algorithm) generates a fixed size 256-bit hash. A hash is a one way function; it can't be decrypted. Above is the hash for the string "password". The following generates a hash:

```
String text = "password";
MessageDigest md = MessageDigest.getInstance("SHA-256");
md.update(text.getBytes("UTF-8"));
byte[] digest = md.digest();
BigInteger bigInt = new BigInteger(1, digest);
String hash = bigInt.toString(16);
```

Refer to `security.HashGeneratorTest`

Configure JNDI name for database connection pool

7. Copy mysql-connector-java to server's glassfish\lib folder
8. Add a MySql JDBC connecti



9. Associate a JNDI name (jdbc/filmstore) with the connection pool

Configure a JDBC Realm

Name	Value
Name	database
JAAS context	jdbcRealm
JNDI	jdbc/filmstore
User Table	usertable
User Name Column	username
Password Column	password
Group Table	grouptable
Group Name Column	groupname
Password Encryption Algorithm	AES
Digest Algorithm	SHA-256

A JDBC realm allows usernames, passwords and groups to be defined dynamically within a set of database tables. JDBC realms are created within Glassfish using the administration console. To create the realm, open the "Configurations | server-config | Security | Realms" tree node. Select the "Realms" node and select the "New..." button displayed in the right hand side panel.

Create a new realm with the name "database". Select JDBC realm in the combo-box on the new realm pane. The class name is "com.sun.enterprise.security.auth.realm.jdbc.JDBCRealm". Several additional properties are required for this realm which describe how the security database is accessed. These additional properties are listed in the table above. JAAS context is the name used to access the LoginModule for authentication.

Mapping users to roles

```
<glassfish-web-app error-url="/error.jsf">
  <security-role-mapping>
    <role-name>admin</role-name>
    <group-name>admin</group-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>customer</role-name>
    <group-name>customer</group-name>
  </security-role-mapping>
</glassfish-web-app>
```

Roles are defined for an application, while groups of users are configured for a particular server. Either add the above security role mapping to sun-web.xml or alternatively, select the Default Principal To Role Mapping checkbox in the security tab of the glassfish admin console. This will associate roles with groups of the same name.

Security configuration using forms authentication

```
<web-app>
   <security-constraint>
   <!-- A list of URL patterns to constrain -->
         <web-resource-collection>
                  <web-resource-name>protected pages</web-resource-name>
                  <url-pattern>/protected/*</url-pattern>
         </web-resource-collection>

   <!-- name the roles authorized to perform the constrained requests -->
         <auth-constraint>
                  <role-name>customer</role-name>
                  <role-name>admin</role-name>
         </auth-constraint>
   </security-constraint>
   <login-config>
         <auth-method>FORM</auth-method>
         <realm-name>database</realm-name>
         <form-login-config>
                  <form-login-page>/login.xhtml</form-login-page>
                  <form-error-page>/error.xhtml</form-error-page>
         </form-login-config>
   </login-config>
</web-app>
```

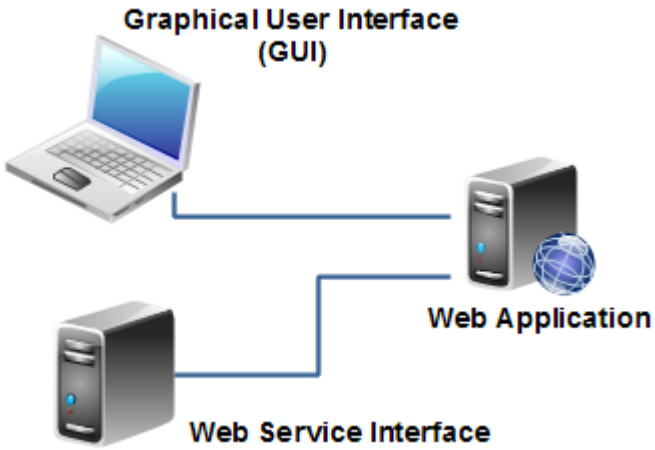
“database” is the JDBC realm configured for the server.

EJB Method Authorisation

- Security identity is propagated from the Web to the EJB container
- Only customer users are authorised to call the addToOrder method
- for multiple roles, pass in an array, for example @RolesAllowed({"customer", "admin"})

```
public class JpaOrderModel {
  @RolesAllowed("customer")
  public String addToOrder(Film film) {
```

REST services



Java API for RESTful Web Services (JAX-RS) is an API that provides support for creating web services according to the Representational State Transfer (REST) architectural pattern.

The annotations include

- @Path specifies the relative path for a resource class or method.
- @GET, @PUT, @POST and @DELETE specify the HTTP request type of a resource.
- @Produces specifies the response Internet media types (used for content negotiation).
    - text/plain
    - application/xml
    - application/json
- @Consumes specifies the accepted request Internet media types.
- @PathParam binds the method parameter to a path segment.

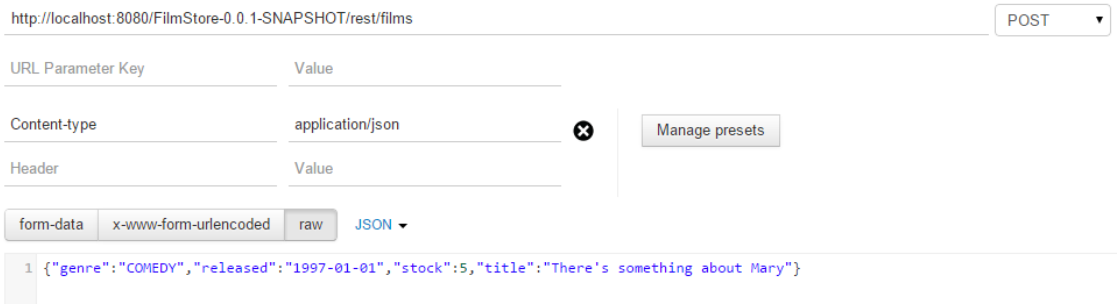
The Application class defines the components of a JAX-RS application and supplies additional meta-data. A JAX-RS application or implementation supplies a concrete subclass of this abstract class. The ApplicationPath annotation identifies the application path that serves as the base URI for all resource URIs provided by Path.

```
@ApplicationPath("/rest")
public class RestConfig extends Application{
}
```

```
import javax.ws.rs.*;
@Path("films")
@Produces(MediaType.APPLICATION_JSON)
public class Service1 {
   private FilmDAO dao = New JpaFilmDAO();
   //URI is http://localhost:8080/Web1/rest/films
   @GET
   public Collection<Film> getAllFilms() {
      return dao.selectAll();
   }
```

```
    //URI is http://localhost:8080/Web1/rest/films/z
    @GET
    @Path("{search}")
    public Collection<Film> getFilmsByTitle(@PathParam("search") String text) {
          return dao.selectByTitle(text);
    }
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createFilm(Film f) {
       dao.insert(f);
       return Response.ok().build();
    }
}
```

An HTTP debugger, such as Fiddler or the chrome postman extension can be used to send HTTP requests to the application.



This is a sample HTTP POST request to the service

```
POST http://simon:8080/FilmStore-0.0.1-SNAPSHOT/rest/films HTTP/1.1
Content-type: application/json
Host: localhost:8080
Content-Length: 91

{"genre":"COMEDY","released":"1997-01-01","stock":5,"title":"There's something
about Mary"}
```

And this is the response

```
HTTP/1.1 200 OK
Server: GlassFish Server Open Source Edition  4.1
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition  4.1
Java/Oracle Corporation/1.8)
Date: Thu, 13 Nov 2014 16:15:41 GMT
Content-Length: 0
```

The HTTP response includes a 3 digit status code. The first digit of the Status-Code defines the class of response:
1xx: Informational - Request received, continuing process
2xx: Success - The action was successfully received, understood, and accepted
3xx: Redirection - Further action must be taken in order to complete the request
4xx: Client Error - The request contains bad syntax or cannot be fulfilled
5xx: Server Error - The server failed to fulfil an apparently valid request

Web service client

There are a number of libraries for encoding and decoding JSON. This is an example of using Google's Gson API. Currently the API can't deserialize dates, so set the Film class's LocalDate field as transient or add a new Film class to the jaxrsClient package with a Date instead of a LocalDate field

```
package jaxrsClient;
public class RestClient {

   private static String urlString =
                                   "http://localhost:8080/IntelliJ-1.0/rest/films/";

   public static void main(String[] args) {
      RestClient client = new RestClient();

      Film film = new Film("There's something about Mary",
                                      2, LocalDate.of(1997, 1, 27), Genre.COMEDY);
      long responseCode = client.insert(film);
      System.out.println(responseCode);

      Collection<Film>films = client.selectByTitle("mary");
      for (Film f : films) {
         System.out.println(f);
      }
   }
   private long insert(Film film) {
      try {
         Gson gson = new GsonBuilder().setDateFormat("yyyy-MM-dd").create();
         String jsonString = gson.toJson(film);
         URL url = new URL(urlString);
         HttpURLConnection connection = (HttpURLConnection) url.openConnection();
         connection.setDoOutput(true);
         connection.setRequestMethod("POST");
         connection.setRequestProperty("Content-Type", "application/json");
         OutputStream os = connection.getOutputStream();
         os.write(jsonString.getBytes());
         os.close();
         connection.disconnect();
         return connection.getResponseCode();
      } catch (IOException e) {
         return 500;
      }
   }
   public Collection<Film> selectByTitle(String search) throws IOException {
      URL url = new URL(urlString+search);
      HttpURLConnection connection = (HttpURLConnection) url.openConnection();
      connection.setRequestMethod("GET");
      connection.setRequestProperty("Accept", "application/json");
      if (connection.getResponseCode() != 200) {
         throw new RuntimeException("Failed : HTTP error code : " +
                                                    connection.getResponseCode());
      }
      BufferedReader reader = new BufferedReader(
                           new InputStreamReader((connection.getInputStream())));
      String jsonString = reader.readLine();
      System.out.println(jsonString);
      connection.disconnect();
      reader.close();
      Gson gson = new Gson();
      Object ob = gson.fromJson(jsonString, Film[].class);
      Film[] films = (Film[]) ob;
      return Arrays.asList(films);
   }
}
```

```
dependencies {
    compile 'com.google.code.gson:gson:2.8.0'
```

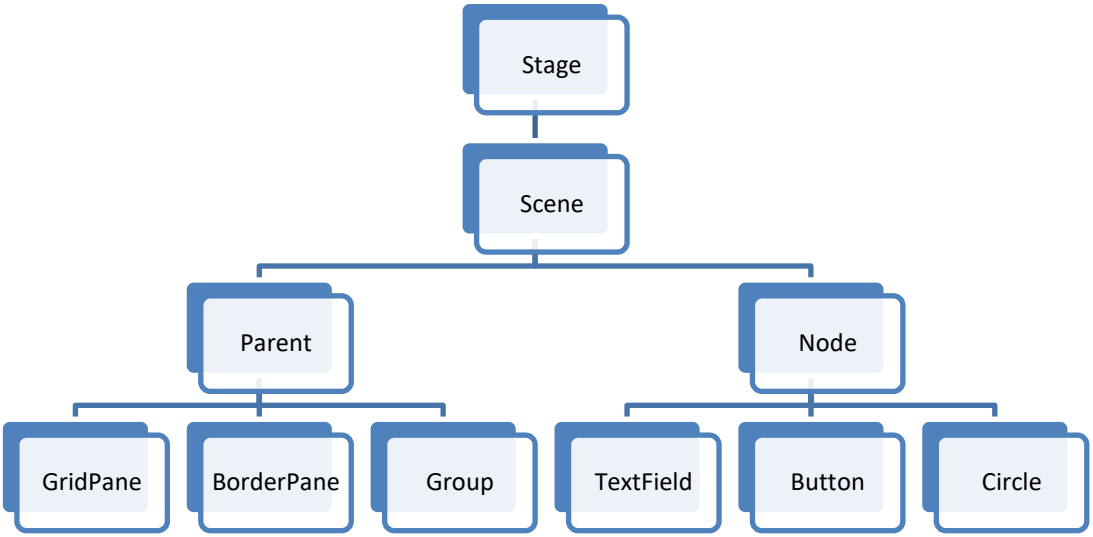
Converting between Date and LocalDate

```
package jaxrsClient;
public class Film {
   private Long id;
   private Genre genre;
   private java.util.Date released;
   private int stock;
   private String title;

   public LocalDate getReleased() {
       return new java.sql.Date(released.getTime()).toLocalDate();
   }

   public void setReleased(LocalDate released) {
       this.released = java.sql.Date.valueOf(released);
   }
```

JavaFX

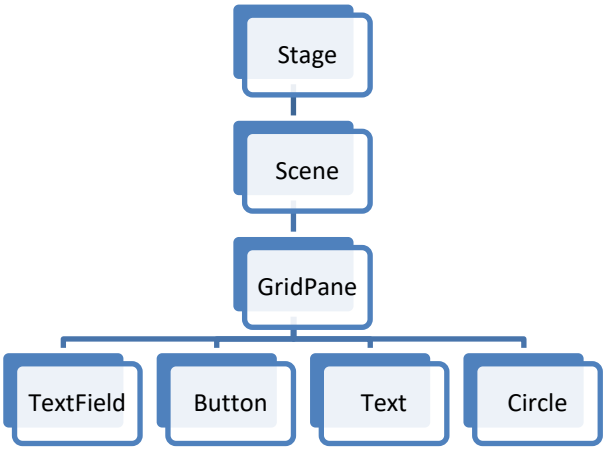
Overview

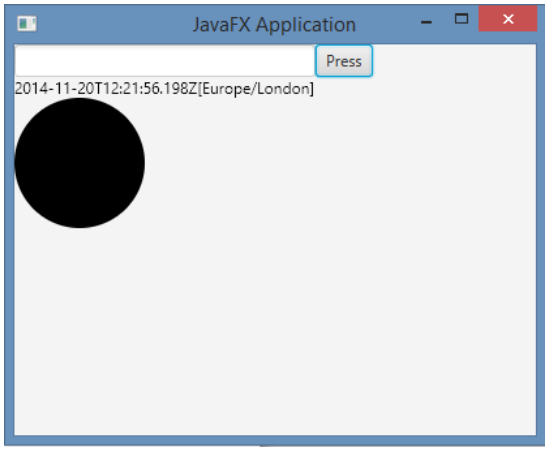
Part of Java 8, JavaFX is a set of graphics and media packages for building “thick client” applications. To add the documentation to the IDE, select File > Project Structure > SDKs and add https://docs.oracle.com/javase/8/javafx/api/



JavaFX applications derive from the Application class and override the start method. The Stage class is the top-level container and the Scene class contains a hierarchical graph of nodes.

```
public class App1 extends Application {
	public void start(Stage stage) {
		// hierarchical graph of nodes
		Button button = new Button("Press");
		TextField textfield = new TextField();
		Text text = new Text();
		Circle circle = new Circle(50);
		//layout classes derive from Pane
		GridPane pane = new GridPane();
		pane.add(textfield, 0, 0);
		pane.add(button, 1, 0);
		pane.add(text, 0, 1);
		pane.add(circle, 0, 2);
		//add the parent node to the scene
		Scene scene = new Scene(pane, 400, 300);
		stage.setTitle("JavaFX Application");
		//add the scene to the Stage; this is the top-level container
		stage.setScene(scene);
		stage.show();
	}
	//main method is not required when using the JavaFX Packager tool
	public static void main(String[] args) {
		launch(args);
	}
}
```





Event handling

The setOnAction method of the Button class takes an EventHandler<ActionEvent> argument. This can be written as an anonymous inner class

```
button.setOnAction(new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent event) {
       text.setText(ZonedDateTime.now().toString());
    }
});
```

or more concisely as a lambda expression.

```
button.setOnAction(event -> text.setText(ZonedDateTime.now().toString()));
```

Properties

Instead of responding to an event, a listener can be added to a property of a component. JavaFX properties implement Property<T>. This is an example of a property named textProperty

```
public class TextField {
   //property sends notifications
   private StringProperty text = new SimpleStringProperty();
   public final StringProperty textProperty() {
      return text;
   }
   public final String getText() {
      return text.get();
   }
   public final void setText(String newValue) {
      text.set(newValue);
   }
```

Binding

Binding enables one property to be automatically updated when another one changes.

```
stage.titleProperty().bind(textfield.textProperty());
```

Binds the titleProperty of the stage object to the textProperty of the textfield object

For more complex bindings, the Bindings class contains static methods to return computed properties. To bind the circle's fill property to a hex value computed from the text property of textfield, use the following expression

```
circle.fillProperty().bind(observableValue);
```

The argument to bind is an ObservableValue<? extends Paint>

```
ObservableValue<Color> observableValue = Bindings.createObjectBinding(
                        callable, // this Callable expression is computed
                        observable // when this observable property changes
);
```

Bindings.createObjectBinding takes two arguments: the function of type Callable<Color> that calculates the value of the binding and the dependencies of the binding. It returns an ObjectBinding, which implements ObservableValue.

```
Callable<Color> callable = ()-> Color.valueOf(textfield.getText());
Observable observable = textfield.textProperty();
```

Including a regular expression to avoid an IllegalArgumentException if the value specifies an invalid hexadecimal value, the binding could be written as a single expression.

```
circle.fillProperty().bind(
        Bindings.createObjectBinding(
                () -> Color.valueOf(textfield.getText()),
                textfield.textProperty()
        )
);
```

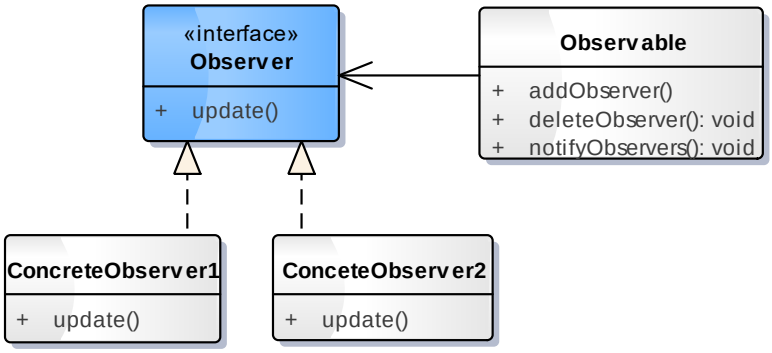
Including validation for the hex colour

```
circle.fillProperty().bind(
        Bindings.createObjectBinding(
                ()-> textfield.getText().matches("[0-9a-fA-F]{6}")?
                     Color.valueOf(textfield.getText()): Color.WHITE,
                     textfield.textProperty()));
```

Observer pattern

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

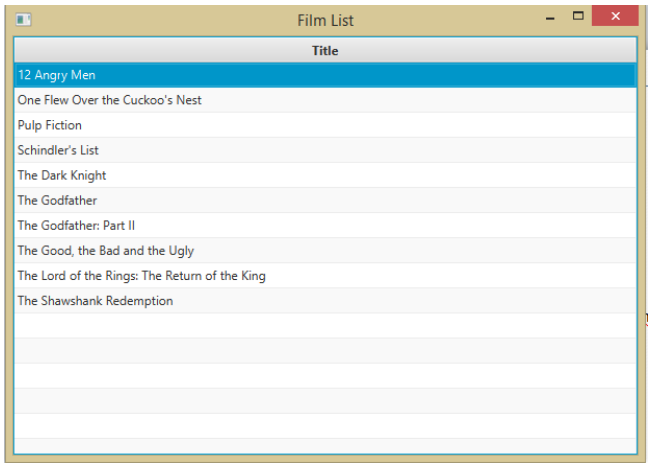
For example textfield.textProperty() is an observable object and circle.fillProperty() is an observer.



OO Design Principles

1. encapsulate what varies
2. favour composition over inheritance
3. program to interfaces not implementations
4. depend on abstractions, not concrete classes
5. classes should be open for extension but closed for modification
6. **strive for loosely coupled designs between objects that interact**

TableView




```
@Override
public void start(Stage primaryStage) {
	// Layout the controls using a BorderPane
	BorderPane pane = new BorderPane();

	// add a TableView to the centre of the BorderPane
	TableView<Film> table = new TableView<>();
	pane.setCenter(table);

	// add a TableColumn
	TableColumn<Film, String> titleCol = new TableColumn<>("Title");
	table.getColumns().add(titleCol);
	titleCol.setPrefWidth(598);

	// The Scene class contains a hierarchical graph of nodes
	Scene scene = new Scene(pane, 600, 400);
	primaryStage.setScene(scene);
	primaryStage.setTitle("Film List");
	primaryStage.show();

	/* PropertyValueFactory<S,T> is a convenience implementation of the
	Callback interface where S is the TableView type and T is the TableColumn
	type. The constructor argument is the property name, extracted reflectively
	from a given TableView row item. */
	titleCol.setCellValueFactory(
				new PropertyValueFactory<Film, String>("title"));

	/*Retrieve a collection of films, create a new observable array list from
	this collection, then pass the ObservableList into the TableView's setData
	method. Changes to the ObservableList will automatically be displayed in
	the TableView.*/
	Collection<Film> films = New JpaFilmDAO().selectAll();
	ObservableList<Film> data = FXCollections.observableArrayList(films);
	table.setItems(data);
}
```

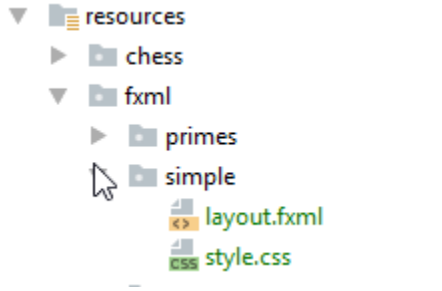
Localisation

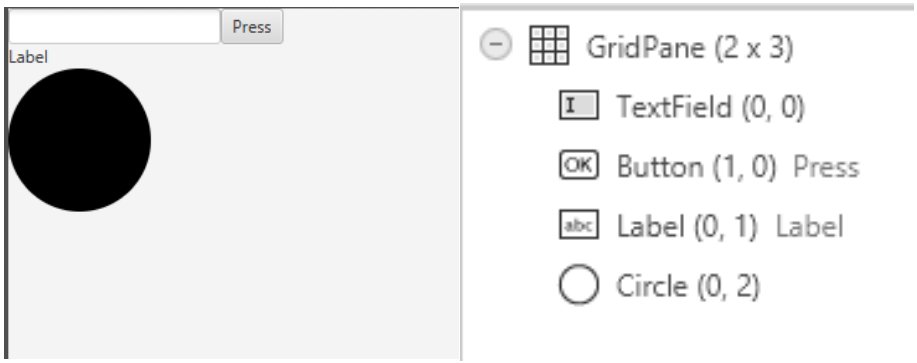
```
//a Locale identifies a language and region
Locale locale1 = Locale.forLanguageTag("es-AR"); //IETF BCP 47
Locale.setDefault(locale1);
System.out.printf(locale1,
  "Numbers and dates formatted for this locale: %-10.2f%,-10d%-15s%tB %4$tY%n",
  Math.PI, 10000, "some text", LocalDate.now());
// Create ResourceBundle from resources.properties
ResourceBundle messages = ResourceBundle.getBundle("resources.labels");
// Fetch the Text from the ResourceBundle
String headerText = messages.getString("header");
System.out.printf("value of header key for this locale: %s", headerText);
```

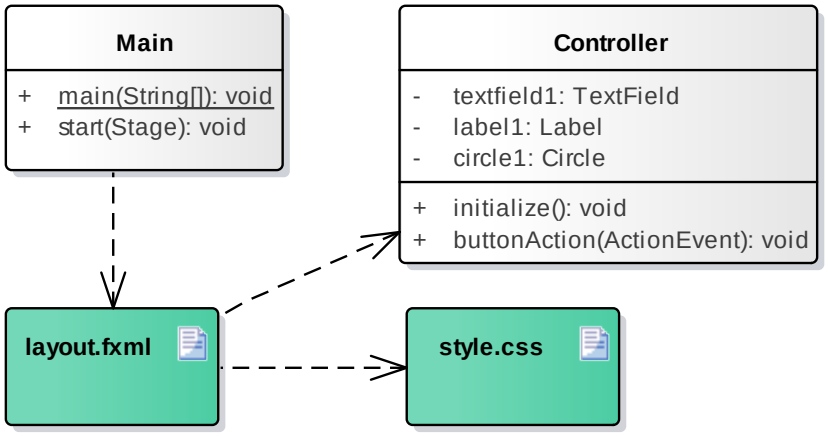
FXML

FXML is a XML-based markup language for defining the user interface of a JavaFX application.

Add a fxml file to the resources folder, add a GridLayout container and two controls, setting their fx:id properties.







/fxml/simple/layout.fxml

```
<GridPane maxHeight="-Infinity" maxWidth="-Infinity"
          xmlns="http://javafx.com/javafx/8.0.112"
          xmlns:fx="http://javafx.com/fxml/1"
          stylesheets="/fxml/simple/style.css"
          fx:controller="fxml.simple.Controller" >
   <children>
      <TextField fx:id="textfield1" />
      <Button fx:id="button1" onAction="#button1Action"
                    text="Press" GridPane.columnIndex="1" />
      <Label fx:id="label1" id="cssStyle" text="Label"
                                       GridPane.rowIndex="1" />
      <Circle fx:id="circle1" radius="50"
                                       GridPane.rowIndex="2" />
   </children>
</GridPane>
```

```
public class Main extends Application {

    public static void main(String[] args) { (args);    }

    @Override
    public void start(Stage primaryStage) {
        try {
            URL url = getClass().getResource(
                                       "/fxml/simple/layout.fxml");
            Pane pane = new FXMLLoader(url).load();
            Scene scene = new Scene(pane);
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Controller {
    //annotation injects the control objects into the controller class
    @FXML
    private TextField textfield1;
    @FXML
    private Label label1;
    @FXML
    private Circle circle1;

    /*
     * FXMLLoader class calls the Controller's initialize() method
     */
    public void initialize() {
        label1.textProperty().bind(textfield1.textProperty());

        Callable<Color> callable = () ->
                    textfield1.getText().matches("[0-9a-fA-F]{6}") ?
                    Color.valueOf(textfield1.getText()) : Color.WHITE;

        StringProperty observable = textfield1.textProperty();
        circle1.fillProperty().bind(Bindings.createObjectBinding(
                                                callable,observable));
    }

    public void button1Action(ActionEvent actionEvent) {
        circle1.setVisible(circle1.isVisible() ? false : true);
    }
}
```

/fxml/simple/style.css

```
/*apply style all TextFields*/
.text-field{
   -fx-background-color: black;
   -fx-font-weight: bold;
   -fx-text-fill: white;
}

/*apply to node with id="#cssStyle"*/
#cssStyle{
    -fx-text-fill: blue;
    -fx-font-size: 16;
    -fx-font-family: monospace;
}
```

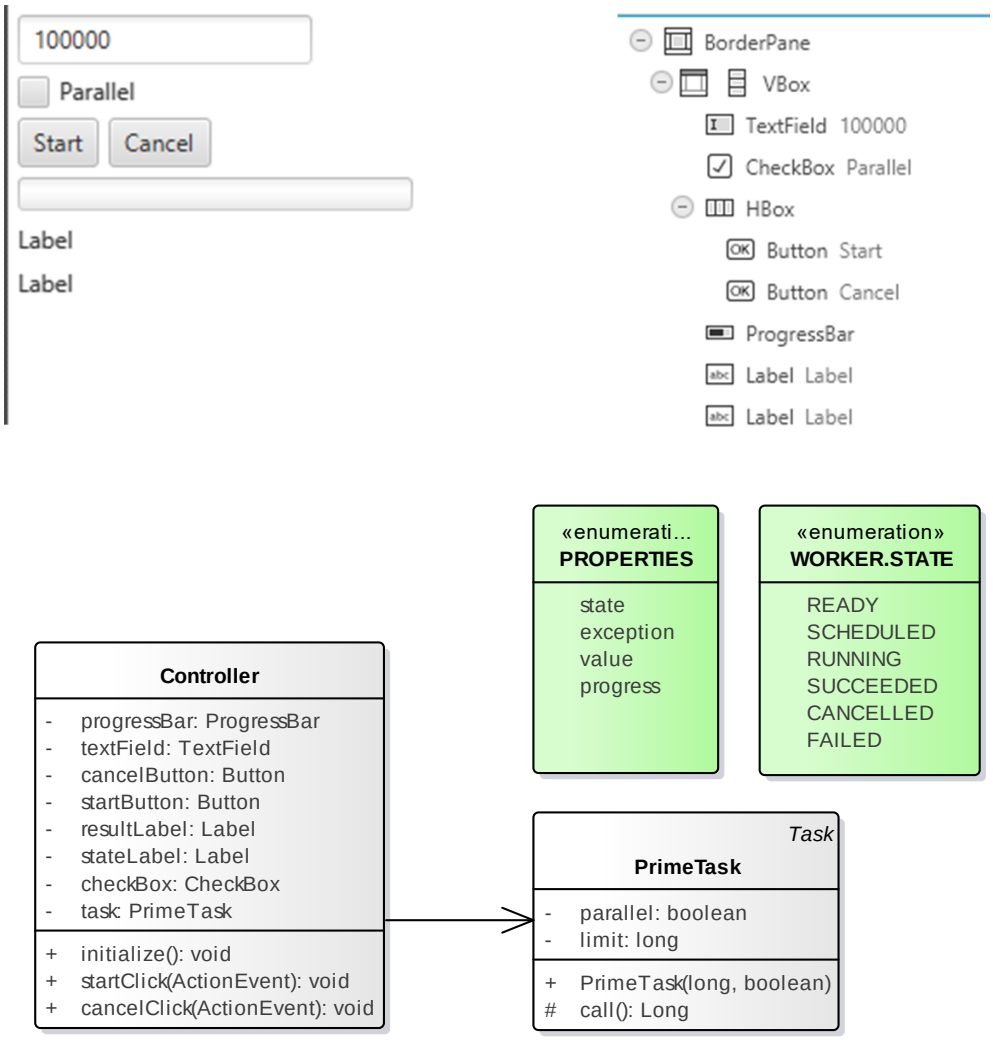
Naming conventions

- CSS style-class
  Separate compound words with hyphens and convert to all lower case. For example, the JavaFX ToggleButton class would have a style-class of "toggle-button".
- CSS property name
  Similar, with the addition of the "-fx-" prefix. For example, the blendMode variable would have a corresponding CSS property name of "-fx-blend-mode".

See JavaFX CSS Reference Guide

Concurrency in JavaFX

Prime number counter



Tasks expose state and observable properties useful for programming asynchronous tasks in JavaFX. An implementation of Task must override the call() method. This method is invoked on the background thread. The Task should check whether it has been cancelled within the body of the call method.

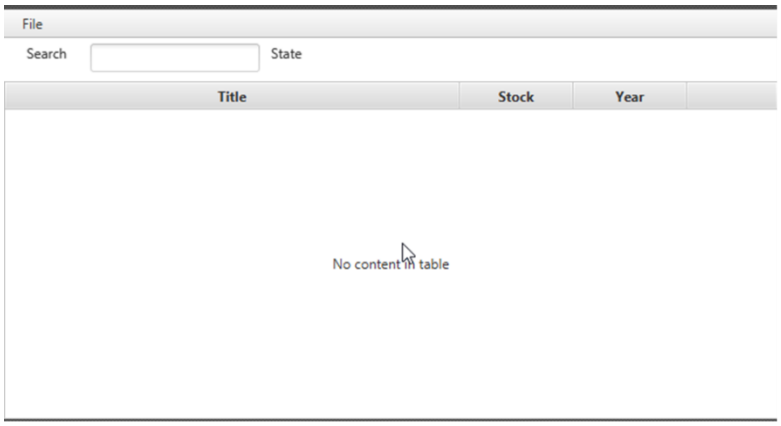
Button handler in Controller class

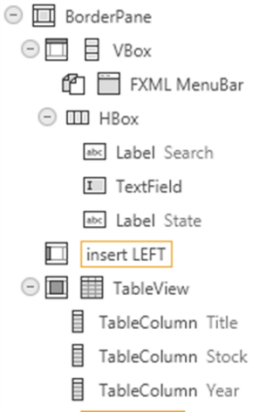
```
public class Controller {
    @FXML    //annotation injects the control objects into the controller class
    ProgressBar progressBar;
    @FXML
    private TextField textField;
    @FXML
    private Button cancelButton;
    @FXML
    private Button startButton;
    @FXML
    private Label resultLabel;
    @FXML
    private Label stateLabel;
    @FXML
    private CheckBox checkBox;

    private PrimeTask task;

    /*
     * An instance of the FXMLLoader class calls the Controller's initialize()
     * method, if available.
     */
    public void initialize() {
    }

    public void startClick(ActionEvent actionEvent) {
        long limit = Long.parseLong(textField.getText());
        task = new PrimeTask(limit, checkBox.isSelected());
        Instant start = Instant.now();

        //bind to progressProperty and stateProperty
        progressBar.progressProperty().bind(task.progressProperty());
        stateLabel.textProperty().bind(task.stateProperty().asString());

        //display result when Task's state transitions to succeeded
        //other properties are onCancelled, onFailed, onRunning, onScheduled
        task.setOnSucceeded(event -> {
            resultLabel.setText(String.format(
                        "%d primes calculated in %d ms, %s",task.getValue(),
                    Duration.between(start,Instant.now()).toMillis(),
                    Thread.currentThread().getName()
            ));
        });

        ExecutorService threadPool = Executors.newSingleThreadExecutor();
        threadPool.execute(task);//start Task in a new thread
    }

    public void cancelClick(ActionEvent actionEvent) {
        task.cancel();
    }
}
```

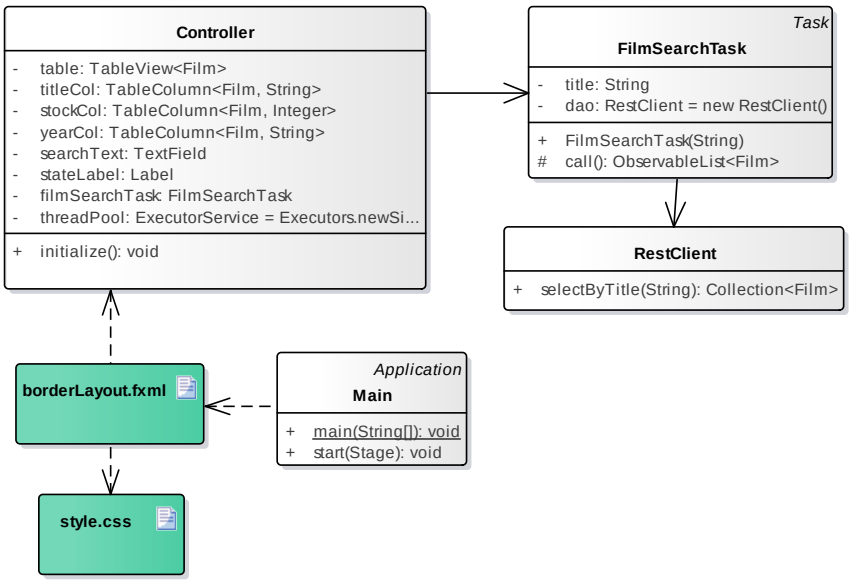
Task class

```
public class PrimeTask extends Task<Long> {
    private long limit;

    public PrimeTask(long limit) {
        this.limit=limit;
    }

    @Override
    protected Long call() throws Exception {
       //runs in worker thread
       System.out.println(Thread.currentThread().getName());
       long count = 0;
       outer:
       for (int i = 2; i < limit; i++) {
           if (isCancelled()) //check if task has been cancelled
               break;
           for (int j = 2; j < i; j++) {
               if (i % j == 0)
                   continue outer;
           }
           count++;
           updateProgress(i, limit);
       }
       return count;
   }
```

Web service client








```
public class FilmSearchTask extends Task<ObservableList<Film>> {
    private String title;
    private RestClient dao = new RestClient();

    public FilmSearchTask(String title) {
        this.title=title;
    }

    @Override
    protected ObservableList<Film> call() throws Exception {
        System.out.printf("Task running in %s%n",
                          Thread.currentThread().getName());
        return FXCollections.observableArrayList(
                              dao.selectByTitle(title));
    }
}
```

```
public class Controller {
   //annotation injects the control objects into the controller class
   @FXML
   private TableView<Film> table;
   @FXML
   private TextField searchText;
   @FXML
   private Label stateLabel;

   private FilmSearchTask filmSearchTask;
   private ExecutorService threadPool =
                              Executors.newSingleThreadExecutor();

   public void initialize() {
      searchText.textProperty().addListener((observable, oldValue,
                                                    newValue) -> {
         if(filmSearchTask != null)
            filmSearchTask.cancel(); //if Task is already running
         filmSearchTask = new FilmSearchTask(newValue);
         stateLabel.textProperty().bind(
                           filmSearchTask.stateProperty().asString());
         threadPool.execute(filmSearchTask);//start Task in a new thread
         //update table when Task's state transitions to succeeded
         //other properties are onCancelled, onFailed, onRunning,
                                                      onScheduled
         filmSearchTask.setOnSucceeded(event ->
                              table.setItems(filmSearchTask.getValue()));
      });
   }
```

Switching scenes

```
public class Controller {

    /*
      Stage
        ^
      Scene
        ^
      Parent (GridPane)
        ^
      Nodes  (Label, Button)
     */

    public void goToScene2Action(ActionEvent actionEvent) throws IOException {
        Button button = (Button) actionEvent.getSource();
        Stage stage = (Stage) button.getScene().getWindow();
        Parent root = FXMLLoader.load(getClass().getResource(
                                        "/fxml/switchScenes/scene2.fxml"));
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
```

fx:include

The <fx:include> tag creates an object from FXML markup defined in another file.

borderlayout.fxml

```
<BorderPane
    <top>
        <VBox BorderPane.alignment="TOP_LEFT">
            <fx:include source="menu.fxml"/>
```

menu.fxml

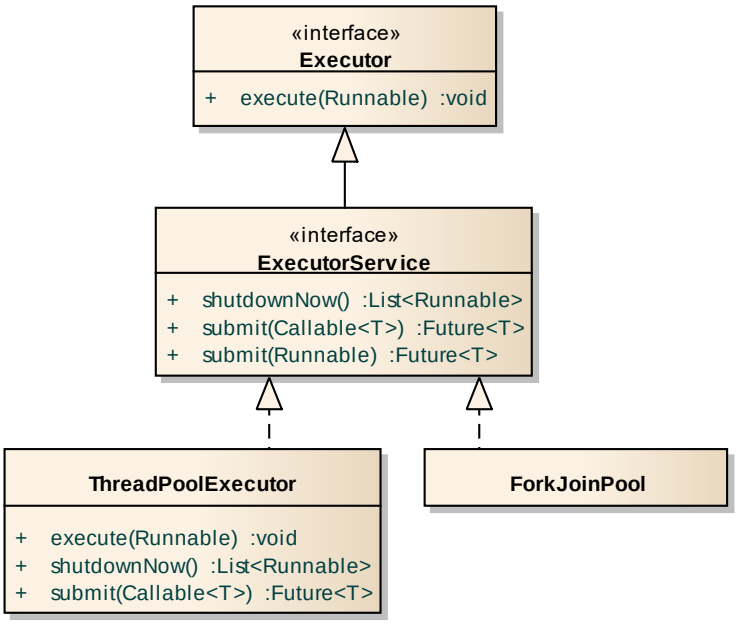
```
<MenuBar xmlns:fx="http://javafx.com/fxml/1"
fx:controller="fxml.table.MenuController" >
    <menus>
        <Menu text="File">
            <items>
                <MenuItem onAction="#handleExitAction"  text="Exit" />
            </items>
        </Menu>
    </menus>
</MenuBar>
```

MenuController.java

```
public class MenuController {
    public void handleExitAction(ActionEvent actionEvent) {
        Platform.exit();
        System.exit(0);
    }
}
```

Concurrency

Executors



Executors simplify threaded programming by starting and managing an application's threads. They can execute Runnable and Callable objects.

Using an Executor

```
public class Blocking {
   public static void main(String[] args) {
      ExecutorService threadPool = Executors.newCachedThreadPool();
      Callable<Long> callable = () ->
             LongStream.range(2, 1000).
             filter(p -> !LongStream.range(2, p).anyMatch(n -> p % n == 0)).
             count();
      Future<Long> future1 = threadPool.submit(callable);
      try {
         long result = future1.get();// blocks until result returned
         System.out.println(result);
      } catch (InterruptedException | ExecutionException e) {
         System.out.println(e.getMessage());
      }
      threadPool.shutdown(); // closes the thread pool
   }
}
```

Thread Pools

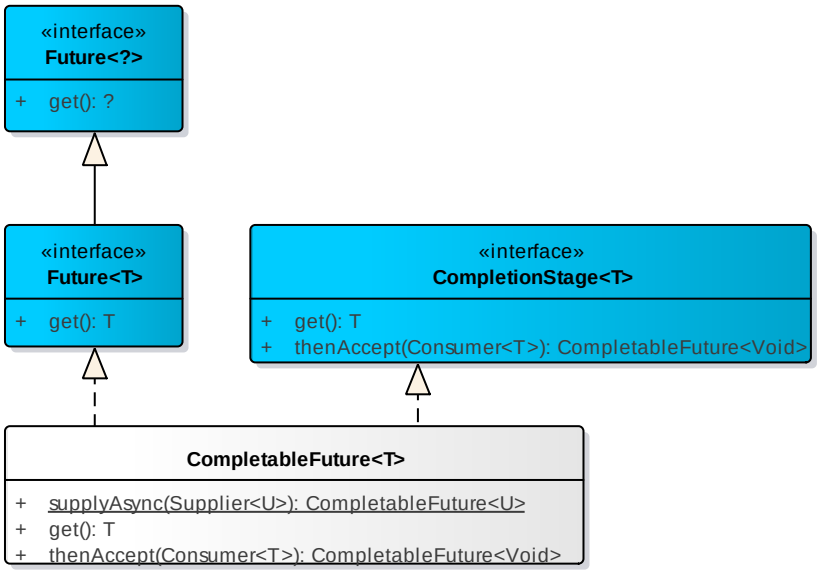
Thread pools consist of worker threads, which exist separately from the Callable tasks that are executed. Using worker threads minimizes the overhead due to thread creation.
A fixed thread pool always has a specified number of threads running; if a thread is terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An expandable thread pool is suitable for applications that launch many short-lived tasks.

The Future interface
A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. To use a Future with a Runnable object, which doesn't return a result, declare the type with an unbounded wildcard: Future<?>.The Callable interface is similar to Runnable, but enables a result to be returned from a method executed in a separate thread.

Asynchronous methods

Although the above prime number calculation is taking place in a separate thread, the get method blocks the main thread until the calculation completes. This can be remedied with CompletionStages. A CompletionStage is a stage of a possibly asynchronous computation that performs an action or computes a value when another CompletionStage completes. The static supplyAsync method returns a new CompletableFuture that is asynchronously completed by a task running in the ForkJoinPool.commonPool() with the value obtained by calling the given Supplier. By calling the thenAccept method of this CompletableFuture, passing in a Consumer argument, the result is displayed in the console.



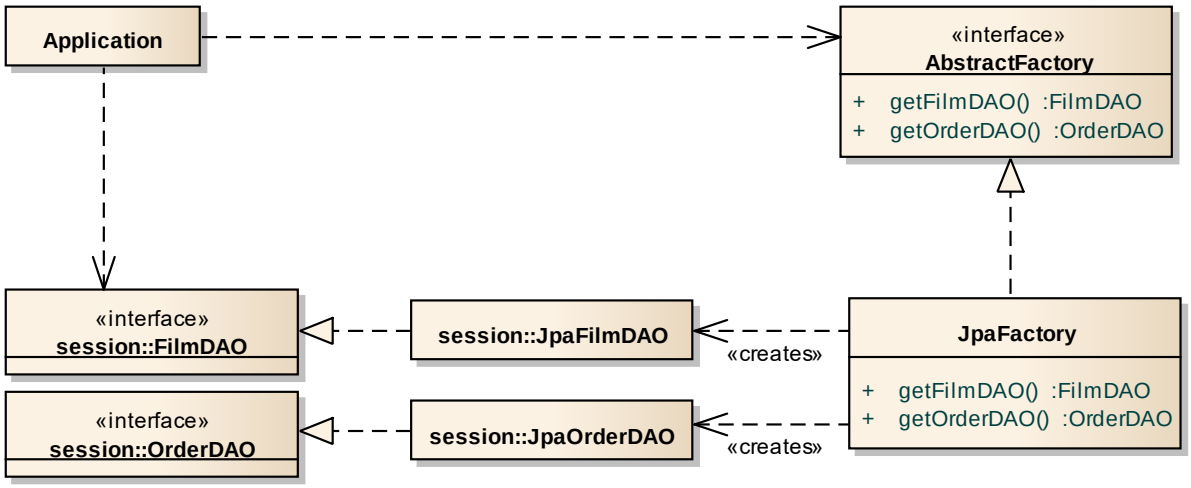

```
Supplier<Long>supplier = () ->
       LongStream.range(2, 1000).
       filter(p -> !LongStream.range(2, p).anyMatch(n -> p % n == 0)).
       count();
Consumer<Long>consumer = n -> System.out.println(n);
CompletableFuture.supplyAsync(supplier).thenAccept(consumer);

//prevents main method exiting
ForkJoinPool.commonPool().awaitTermination(5, TimeUnit.SECONDS);
```

Categorising design patterns

- **Creational** - class instantiation
  - Abstract factory
  - Factory method
  - Singleton
- **Structural** - class and object composition
  - Observer
  - Command
- **Behavioural** – communication between objects
  - Strategy
  - Decorator
  - Facade

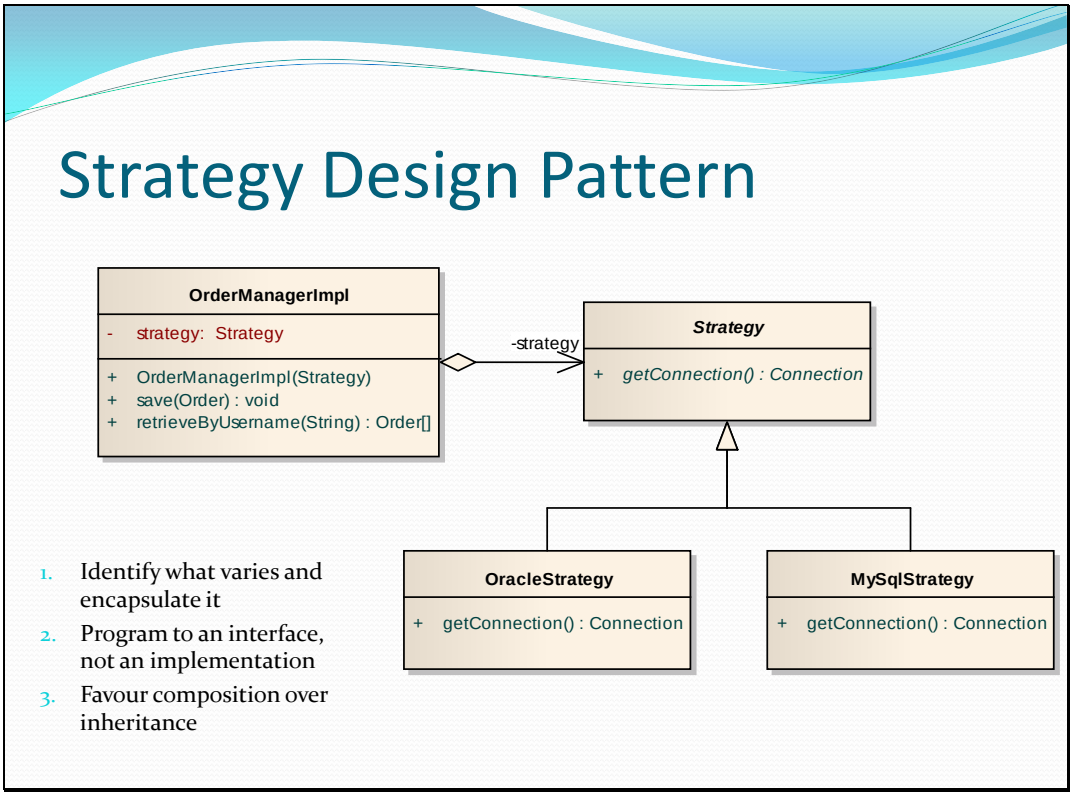
Abstract Factory

The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme




```
public interface AbstractFactory {
   FilmDAO getFilmDAO();
   OrderDAO getOrderDAO();
}
public class JpaFactory implements AbstractFactory {
   @Override
   public FilmDAO getFilmDAO() {
      return new JpaFilmDAO();
   }
   @Override
   public OrderDAO getOrderDAO() {
      return new JpaOrderDAO();
   }
}
public class FactoryMaker {
   public static AbstractFactory getFactory(String key) {
      switch (key) {
      case "jpa":
         return new JpaFactory();
      case "jdbc":
         // return new JdbcFactory();
      default:
         return null;
      }
   }
}
public class Application {
   public static void main(String[] args) {
      AbstractFactory af = FactoryMaker.getFactory("jpa");
      FilmDAO filmDAO = af.getFilmDAO();
      filmDAO.selectAll();
```

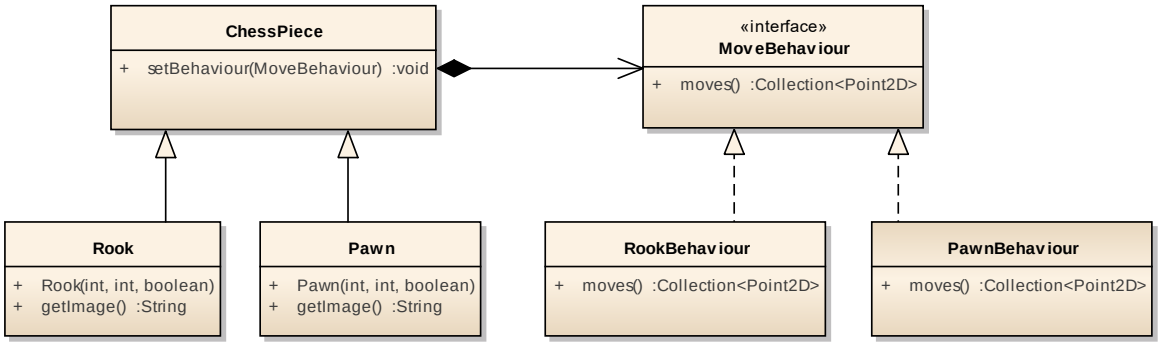
```
        OrderDAO orderDAO = af.getOrderDAO();
        orderDAO.persistOrder();
    }
}
```

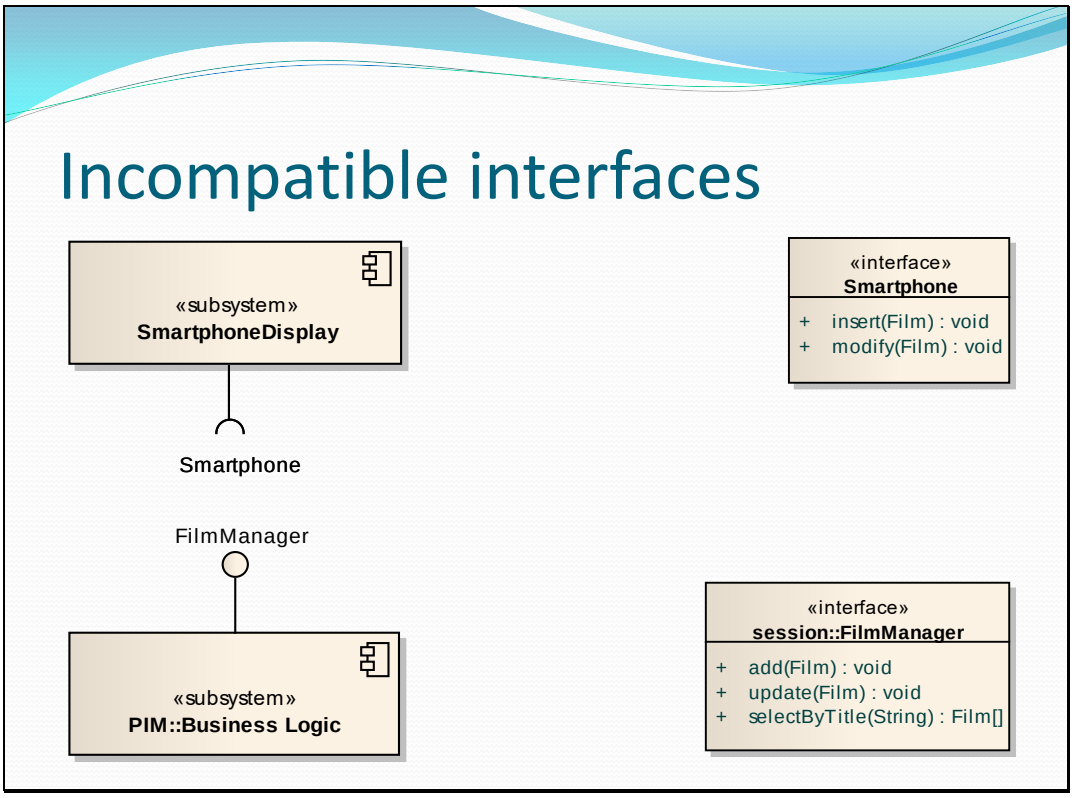


- Identify what varies and encapsulate it, so that later you can alter or extend the parts that vary without affecting those that don't. The connection behaviour of the OrderManagerImpl class is encapsulated in an abstract class called Strategy, with implementations for a number of databases.
- Program to an interface or abstract class, not an implementation, to exploit polymorphism.
- Favour composition over inheritance. This enables behaviour to be changed at runtime.

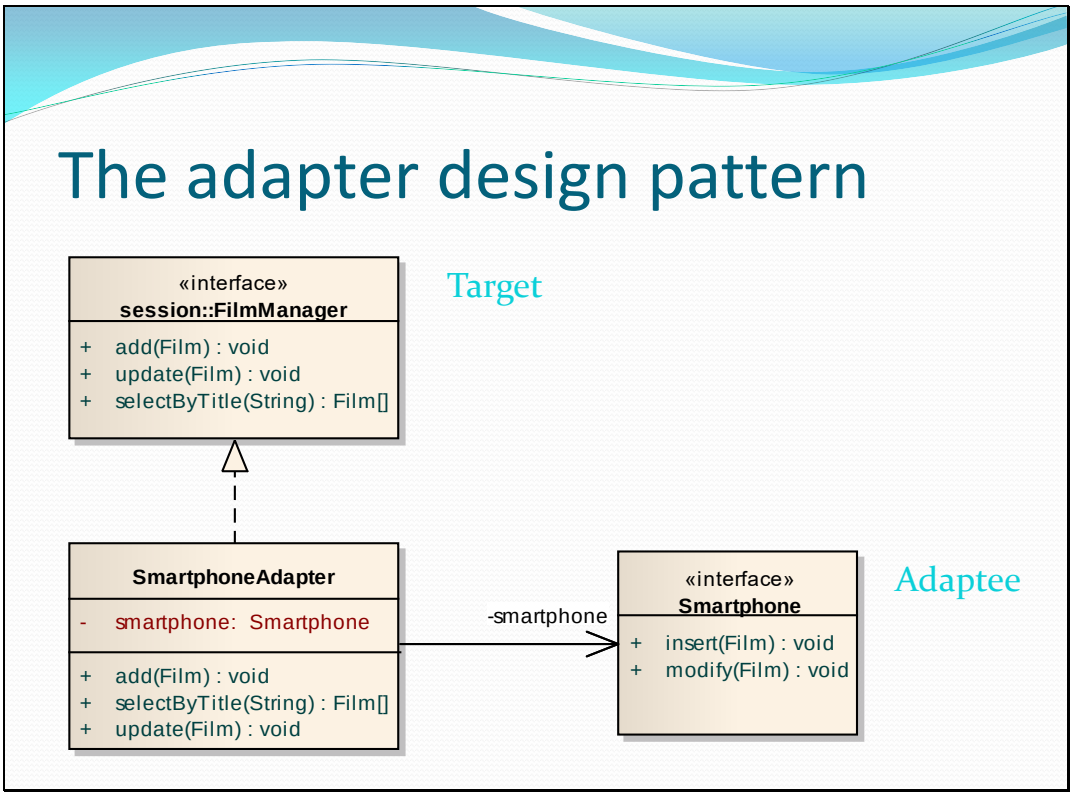
Using the Strategy Design Pattern

```
public class OrderManagerImpl {

    private Strategy strategy;

    public OrderManagerImpl(Strategy strategy) {
        this.strategy = strategy;
    }

    public void save(Order order){
        Connection c = strategy.getConnection();
    }
}
```





A different company has developed a SmartphoneDisplay component which connects to the business logic layer through a required interface called Smartphone. Since this interface is incompatible with the FilmManager interface, the adapter design pattern can be used to connect the two components.

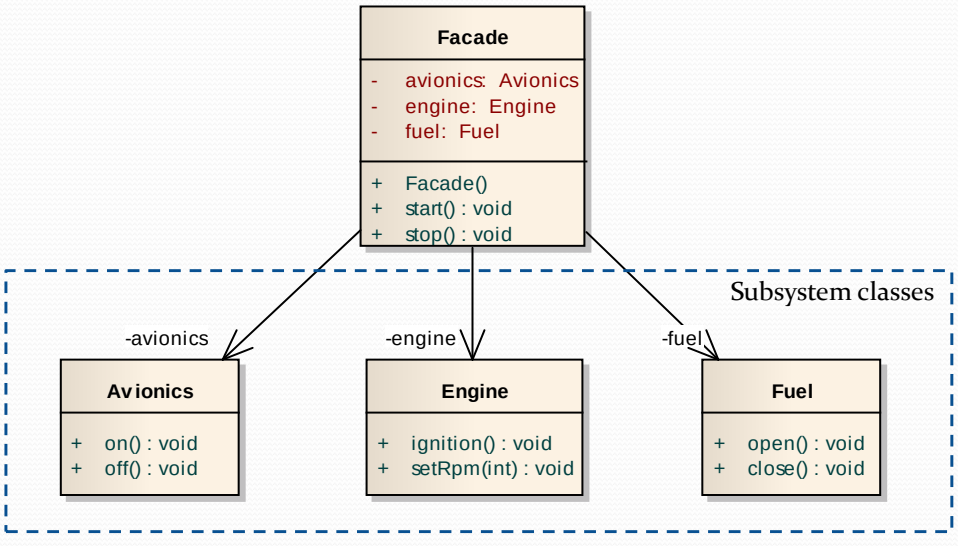


The adapter design pattern converts the interface of a class into another interface that clients expect.

The Adapter class

```
public class SmartphoneAdapter implements FilmManager {

    private Smartphone smartphone;

    @Override
    public void add(Film film) {
        smartphone.insert(film);
    }

    @Override
    public void update(Film film) {
        smartphone.modify(film);
    }
```

The Facade Design Pattern



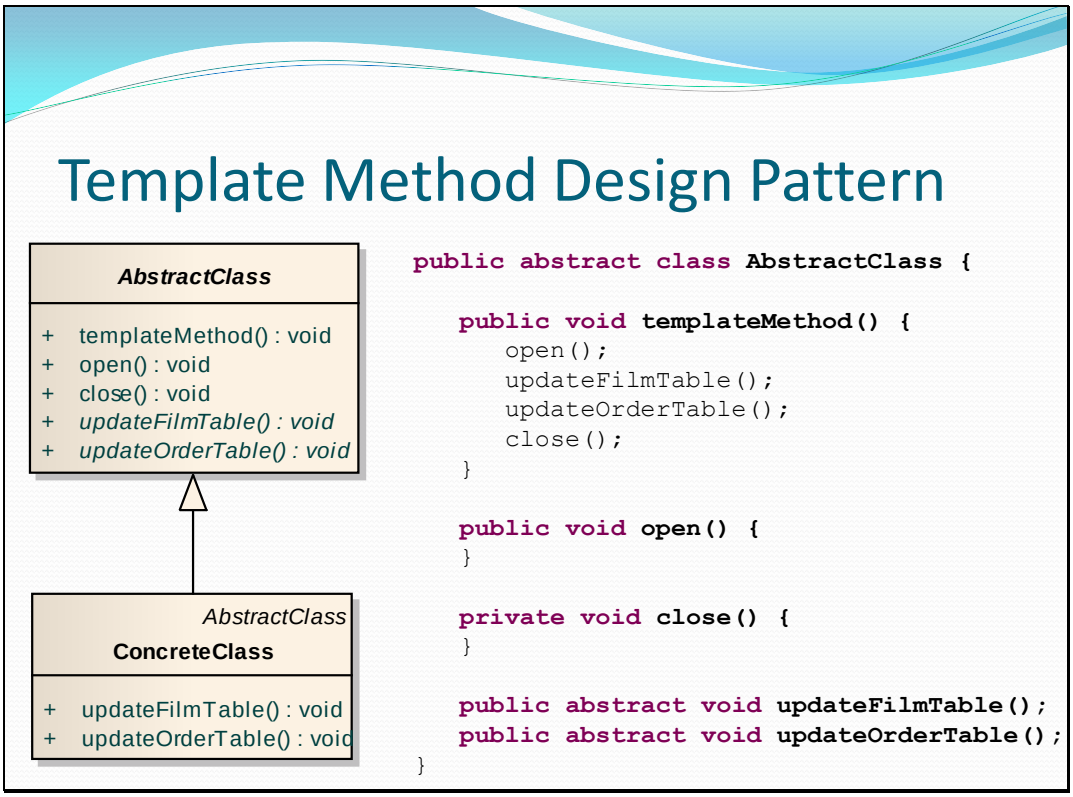
This pattern provides a unified interface to a set of interfaces in a subsystem.

The Facade class

```
public class Facade {
    private Avionics avionics = new Avionics();
    private Engine engine = new Engine();
    private Fuel fuel = new Fuel();

    public void start(){
        fuel.open();
        engine.setRpm(1200);
        engine.ignition();
        avionics.on();
    }

    public void stop(){
        engine.setRpm(1200);
        fuel.close();
        avionics.off();
    }
}
```

The above façade simplifies interaction with the subsystem.

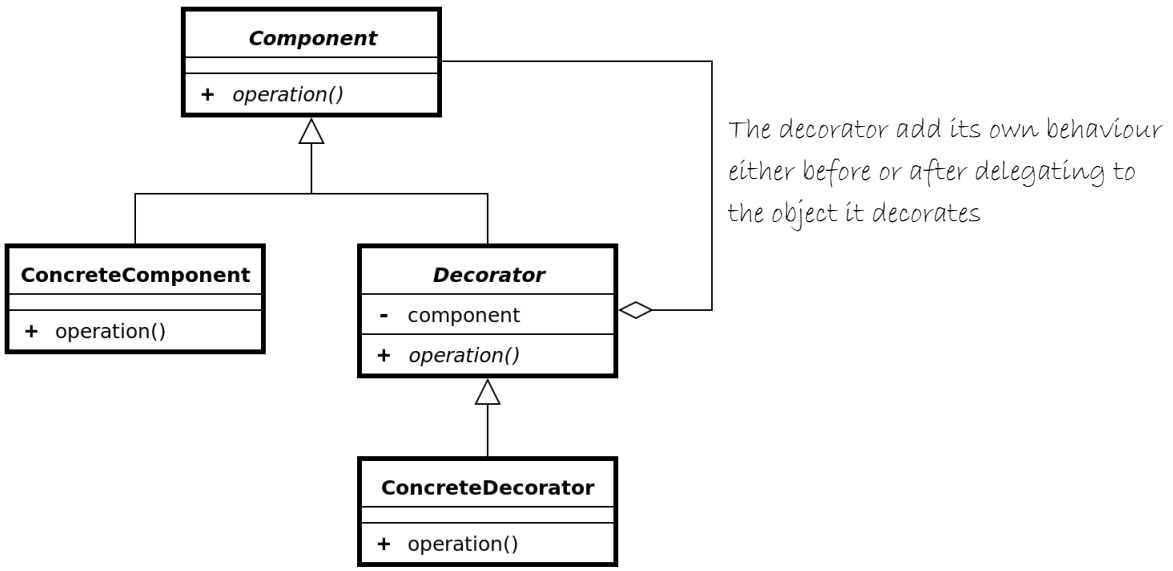
The Facade Pattern follows the Principle of Least Knowledge, or in other words “talk only to your immediate friends”. This reduces dependencies between objects, which can reduce software maintenance. It is a principle rather than a law, so can sometimes be ignored (eg System.out.println).



Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behaviour. In the above example, templateMethod defines a skeleton algorithm for purchasing films, in which the implementation of the two abstract methods is deferred to a derived class.

Decorator pattern

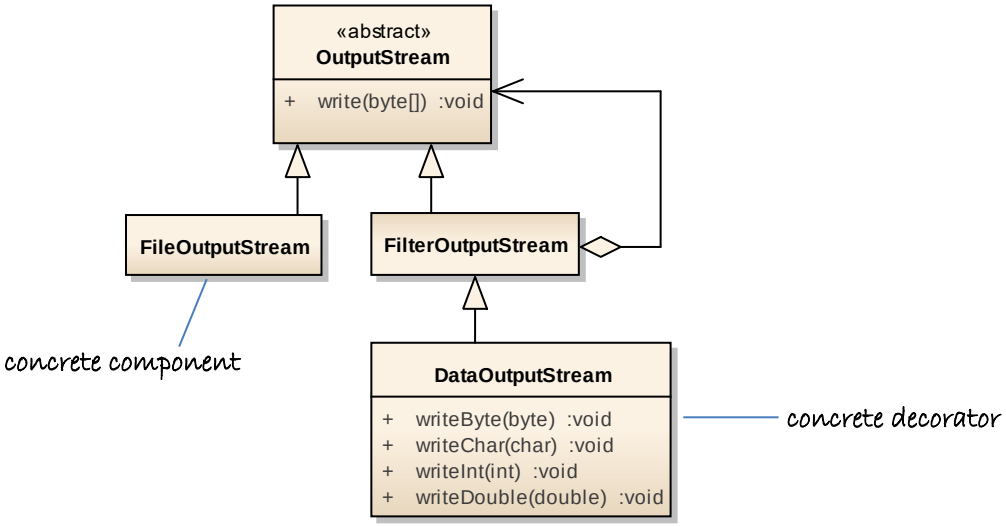
The decorator pattern attaches additional responsibilities to an object dynamically.



OO Design Principles

1. encapsulate what varies
2. favour composition over inheritance
3. program to interfaces not implementations
4. depend on abstractions, not concrete classes
5. **classes should be open for extension but closed for modification**

FileOutputStream



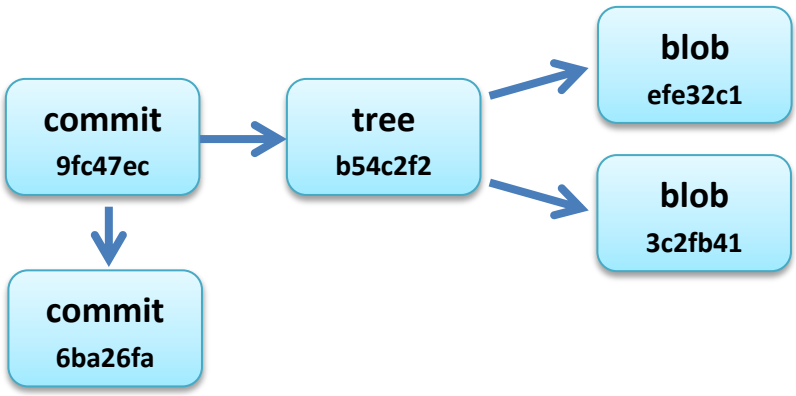
Git is a distributed revision control system with an emphasis on speed, data integrity and support for distributed, non-linear workflows.

Every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.

Workflow

1. modify files in the working directory
2. add files to the index
   i. SHA-1 checksum is generated for each file
   ii. the bytes are stored in the repository as a blob
   iii. >git add *
   iv. >git status

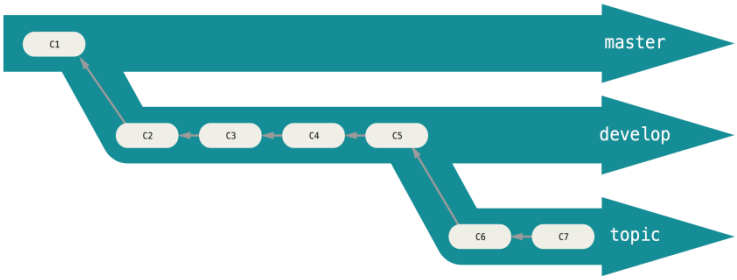
3. commit files in the index; this creates a snapshot of the project
   i. tree object with checksum is generated for project directories
   ii. commit object with checksum includes metadata and points at root directory and previous commit
   iii. >git commit –m “message”



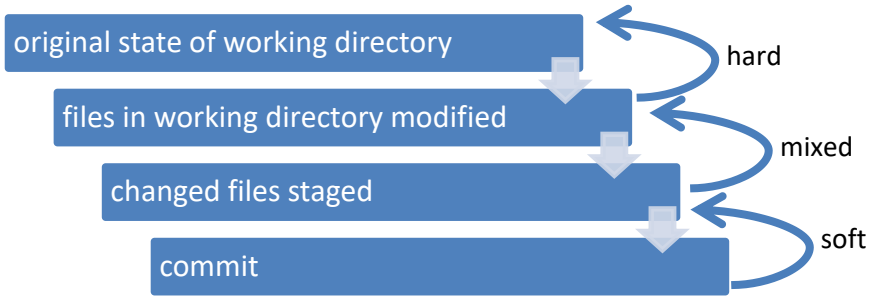
Branching

A branch is a pointer to a commit. The default branch name in Git is **master**.
A pointer called **HEAD** tracks the current branch.

A workflow might maintain a master branch for stable code that will be released; a branch named develop to test stability and short-lived topic branches

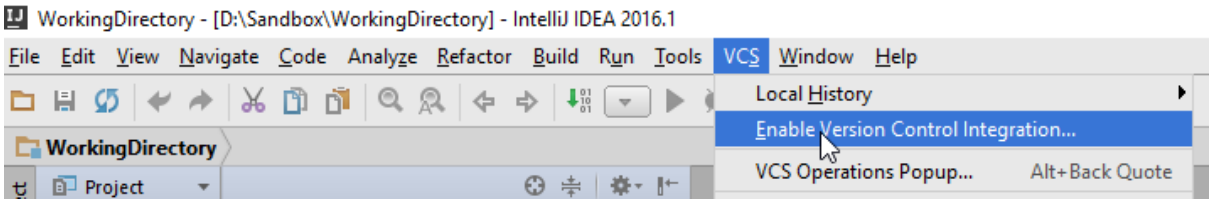


git reset [<mode>] [<commit>] resets the current branch head to <commit> and possibly updates the index (resetting it to the tree of <commit>) and the working tree depending on <mode>. The mode can be soft, mixed or hard.



Create a local and repository

1. Create a local repository in IntelliJ



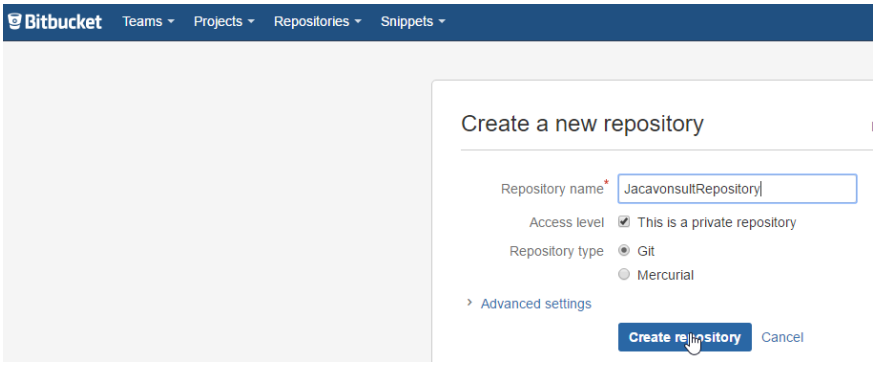
2. Add a .gitignore file (see https://www.gitignore.io/)

```
# Intellij project files
.idea/
#Gradle
.gradle/
build/
```

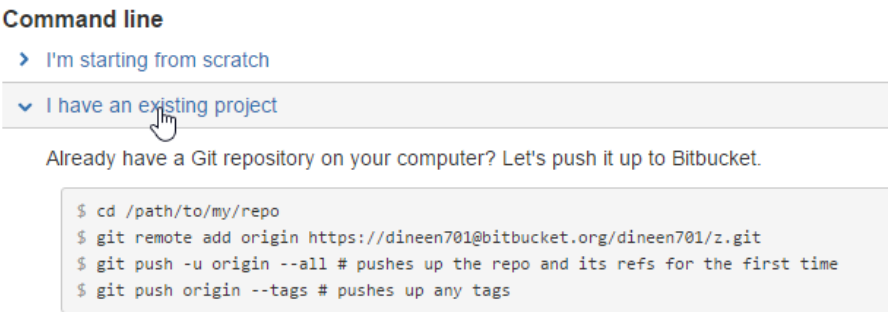
3. Open the working directory in source tree
4. Commit in master branch
5. Add develop branch

Push to a remote repository

1. Log in to Biitbucket, select the Repositories menu and create a new repository



2. Select "I have an existing project" and copy the commands



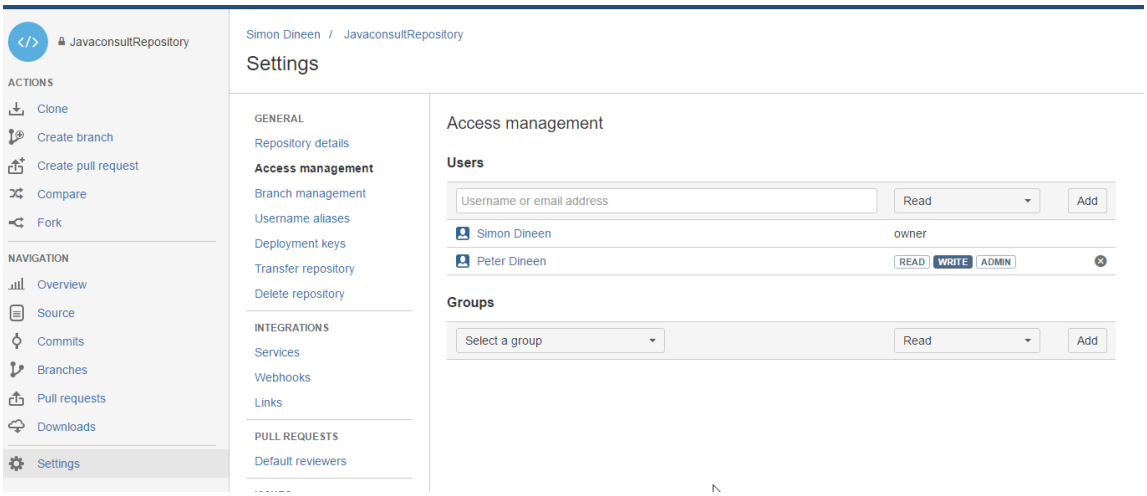
3. To add a new remote, open the source tree console and use the git remote add command. This takes two arguments; a remote name, for example, origin, and a remote URL, for example, https://dineen701@bitbucket.org/dineen701/java-course.git

```
User@OFFICE /d/Sandbox/WorkingDirectory (develop)
$ git remote add origin https://dineen701@bitbucket.org/dineen701/javaconsultre
pository.git

User@OFFICE /d/Sandbox/WorkingDirectory (develop)
$ git push -u origin --all # pushes up the repo and its refs for the first time
Password for 'https://dineen701@bitbucket.org':
Counting objects: 30, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (30/30), 51.83 KiB | 0 bytes/s, done.
Total 30 (delta 4), reused 0 (delta 0)
To https://dineen701@bitbucket.org/dineen701/javaconsultrepository.git
 * [new branch]      develop -> develop
 * [new branch]      master -> master
Branch develop set up to track remote branch develop from origin.
Branch master set up to track remote branch master from origin.
```

4. Push both branches to the remote, either by running “git push origin --all" or by clicking the Push button

Share the repository

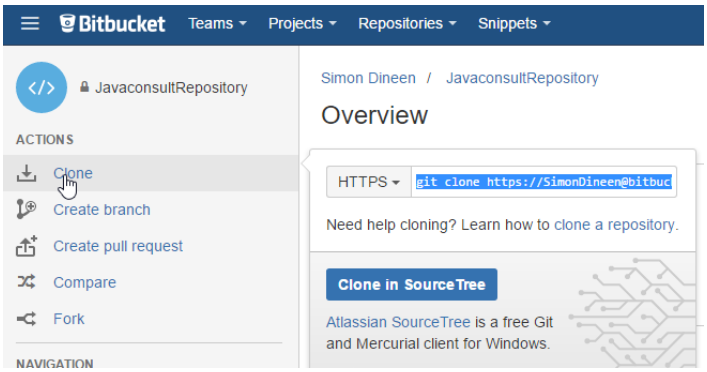
a. select repository in Bitbucket
b. click Settings button on left
c. select Access management



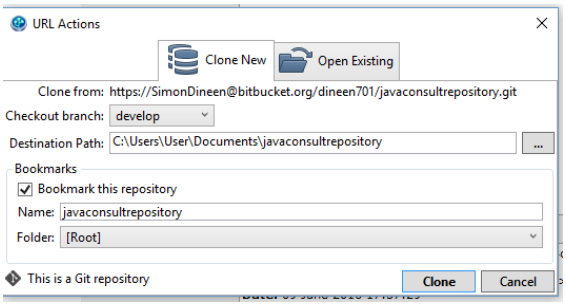
5. type username, select Write access

Cloning the repository

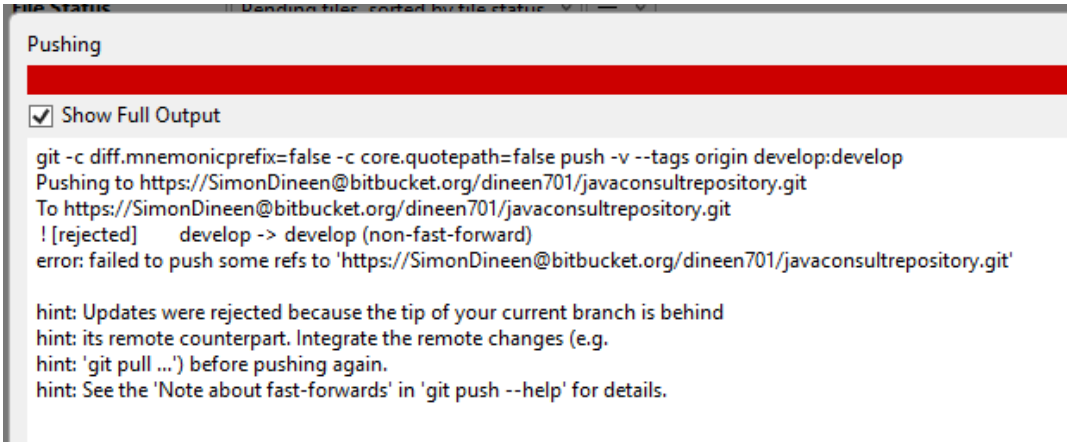
6. Another user logs in to bitbucket
7. Click “Clone in sourcetree”



8. Checkout develop branch



9. Open project in intellij (see destination path above)
10. Edit the working directory
11. Commit and push
12. The other developer pulls the branch from source tree
13. If a developer pushes a commit to the remote repository, another developer attempting to push before pulling will encounter an error



Solutions

InMemoryFilmDAO

```
public class InMemoryFilmDAO {
	private Long currentId = 1L;
	private Map<Long, Film> films = new HashMap<>();

	@Override
	public boolean delete(Long filmId) {
		boolean deleted =  films.remove(filmId) == null ? false : true;
		return deleted;
	}

	@Override
	public Long insert(Film film) {
		Long id = currentId++;
		film.setId(id);
		films.putIfAbsent(id, film);
		return id;
	}

	@Override
	public Collection<Film> selectAll() {
		return films.values();
	}


	@Override
	public Film selectById(Long id) {
		return films.get(id);
	}

	@Override
	public Collection<Film> selectByTitle(String search) {
		return null;
	}

	@Override
	public boolean update(Film film) {
		boolean updated = films.replace(film.getId(), film) == null ?
                                                                false : true;
		return updated;
	}
}
```

JavaFX asynchronous prime number counter

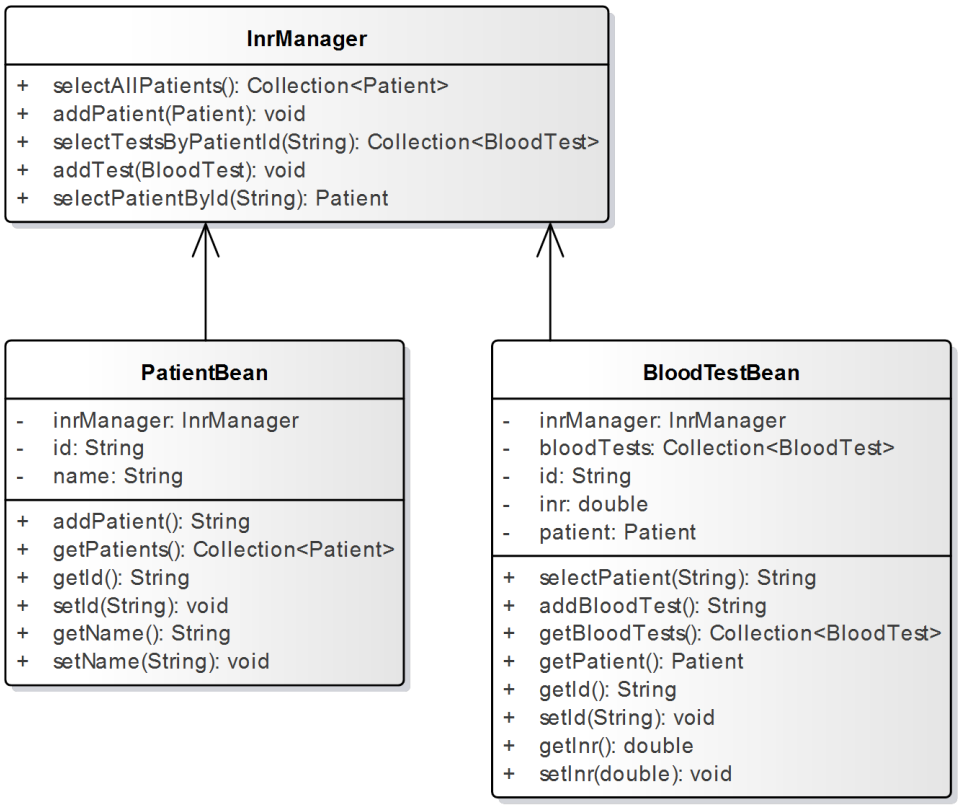
The above console application can be written as a desktop application. UI components can't be updated from a separate thread: the static runLater method posts the Runnable argument to the event queue and then returns immediately to the caller.

```
public class JavaFX extends Application {
   public void start(Stage stage) {
       Button button1 = new Button("Press");
       TextField textfield = new TextField("10000000");
       Text text1 = new Text();
       FlowPane pane = new FlowPane();
       pane.setHgap(10);
       pane.getChildren().add(textfield);
       pane.getChildren().add(button1);
       pane.getChildren().add(text1);

       Scene scene = new Scene(pane, 400, 300);
       stage.setTitle("JavaFX Application");
       stage.setScene(scene);
       stage.show();

       button1.setOnAction(event -> {

          //calculated synchronously on UI thread
          //text1.setText(primeCount(textfield.getText()));

          //calculated asynchronously
          Supplier<String> supplier = () -> primeCount(textfield.getText());

          //displays the result
          Consumer<String> consumer = s ->
                          Platform.runLater(() -> text1.setText(s));

          CompletableFuture.supplyAsync(supplier).thenAccept(consumer);

          text1.setText("calculating...");
       });
   }

   public static String primeCount(String s) {
       long limit = Long.parseLong(s);
       return LongStream.range(2, limit).
                   parallel().
                   filter(p -> !IntStream.range(2, (int) Math.sqrt(p) + 1).
                   anyMatch(n -> p % n == 0)).
                   count()+ " prime numbers";
       }

       public static void main(String[] args) {
             launch(args);
       }
}
```

INR

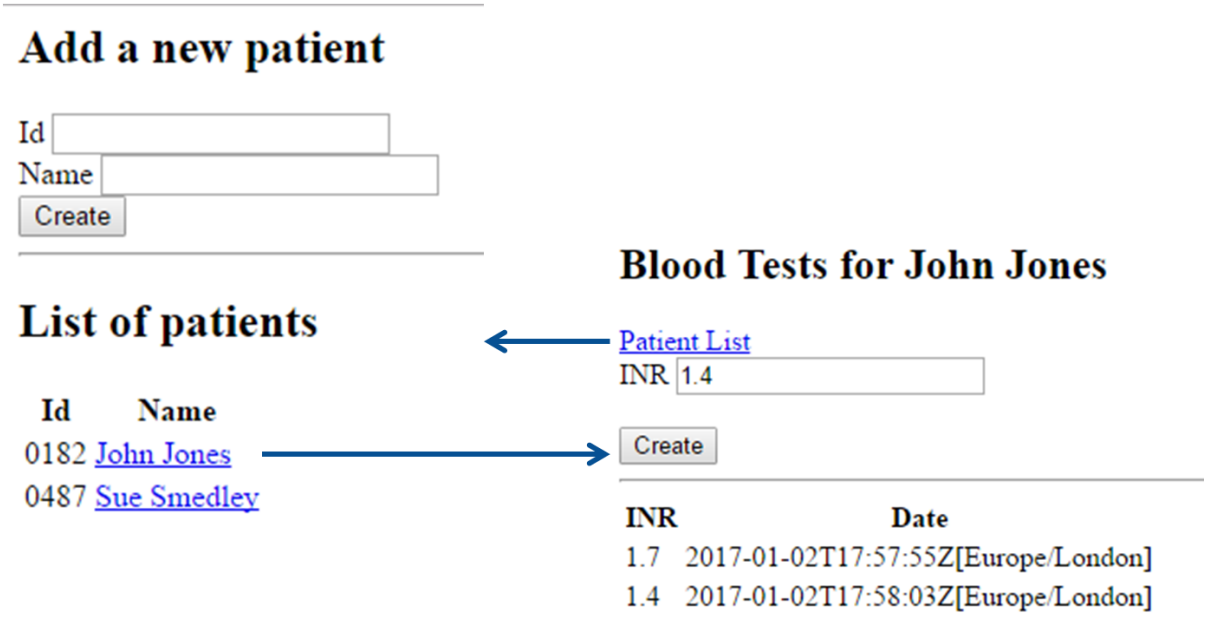
INR is based on the ratio of a patient's blood clotting time to average clotting time. Normal values are 0.8 to 1.1, or for patients on blood thinning medication, 2.0 to 3.0.

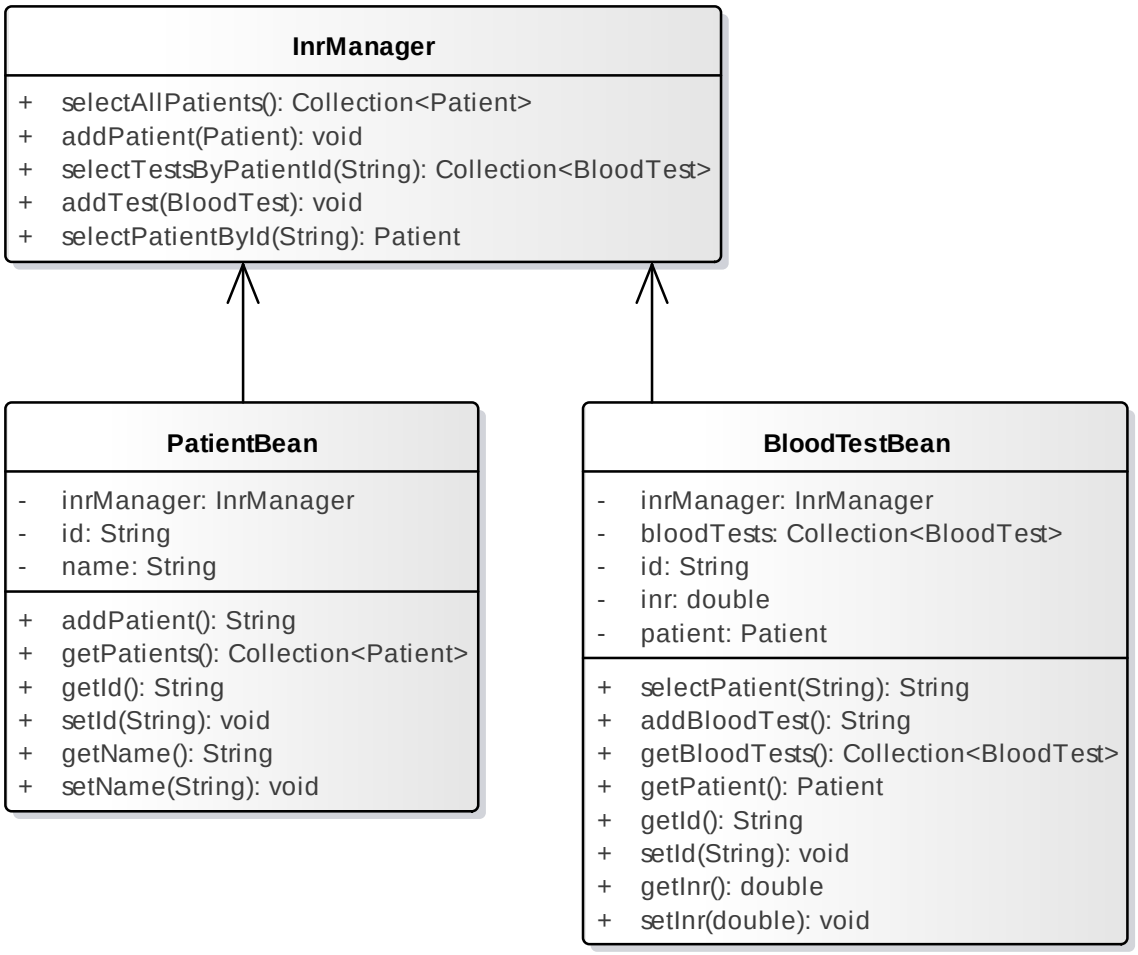
JPA



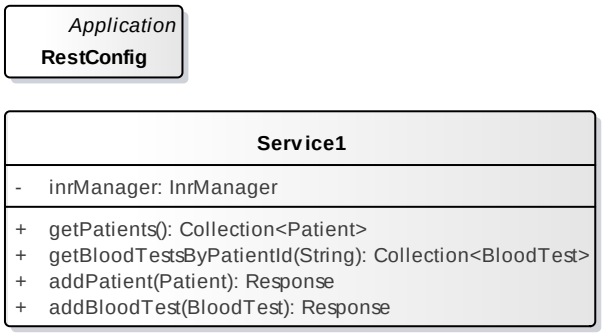
patientsView.xhtml

bloodTestsView.xhtml




InrManager
+ selectAllPatients(): Collection<Patient>
+ addPatient(Patient): void
+ selectTestsByPatientId(String): Collection<BloodTest>
+ addTest(BloodTest): void
+ selectPatientById(String): Patient
PatientBean
- inrManager: InrManager
- id: String
- name: String
+ addPatient(): String
+ getPatients(): Collection<Patient>
+ getId(): String
+ setId(String): void
+ getName(): String
+ setName(String): void
BloodTestBean
- inrManager: InrManager
- bloodTests: Collection<BloodTest>
- id: String
- inr: double
- patient: Patient
+ selectPatient(String): String
+ addBloodTest(): String
+ getBloodTests(): Collection<BloodTest>
+ getPatient(): Patient
+ getId(): String
+ setId(String): void
+ getInr(): double
+ setInr(double): void

REST


Application
RestConfig
Service1
- inrManager: InrManager
+ getPatients(): Collection<Patient>
+ getBloodTestsByPatientId(String): Collection<BloodTest>
+ addPatient(Patient): Response
+ addBloodTest(BloodTest): Response

Channels and ByteBuffers

In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.

Buffer

A linear, finite sequence of elements of a specific primitive type.

- Capacity – the number of elements it contains. Never changes.
- Limit – the index of the first element that should not be read or written.
- Position – the index of the next element to be read or written.

Position <= Limit <= Capacity

Write from a buffer to a channel

```
Path path = Paths.get("C:/Users/User/Documents/file.txt");
WritableByteChannel channel = Files.newByteChannel(path,
EnumSet.of(CREATE, WRITE));
String text = "abcde";
//capacity and limit will be array.length; position zero; mark
undefined
ByteBuffer byteBuffer = ByteBuffer.wrap(text.getBytes());
//Writes a sequence of bytes from buffer to channel
channel.write(byteBuffer);
```

Read from a channel into a buffer

```
SeekableByteChannel sbc = Files.newByteChannel(path);
// Allocate a new byte buffer with position 0, limit=capacity = 4
ByteBuffer buf = ByteBuffer.allocate(4);
int bytesRead = 0;
//Read a sequence of bytes from the channel into the buffer
while ((bytesRead = sbc.read(buf))!= -1) {
    buf.flip();//sets limit to position
    //elements between the current position and limit
    while (buf.hasRemaining()) {
        //read the byte at buffer's current position,
        //then increment position.
        System.out.print((char) buf.get()); // read 1 byte at a time
    }
    buf.position(0);
}
```

0	1	2	3	position	limit	capacity
0	0	0	0	0	4	4

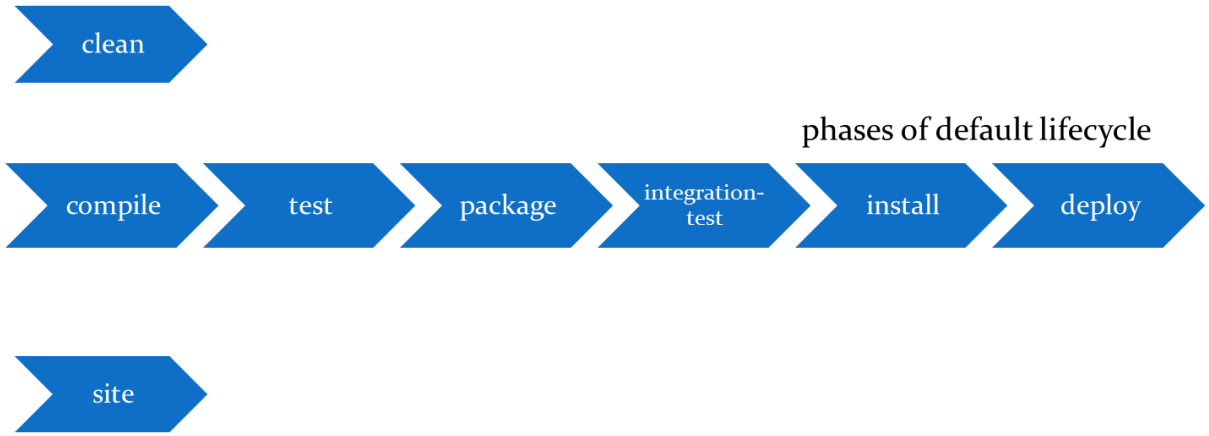
Maven

http://books.sonatype.com/mvnref-book/reference/index.html

For the majority of users, Maven is a build tool that compiles, tests and packages deployable artifacts from source code.
Maven's adoption of convention over configuration specifies default directory locations, a defined life-cycle and a set of common plugins that know how to build and assemble software. If you follow the conventions, Maven will require almost zero effort - just put your source in the correct directory, and Maven will take care of the rest.

The Build Lifecycle

A lifecyle is an organized sequence of phases that exist to give order to a set of goals. There are three standard lifecycles in Maven: clean, default and site.
http://books.sonatype.com/mvnref-book/reference/lifecycle.html



The specific goals bound to each phase default to a set of goals specific to a project's packaging. A project with packaging jar has a different set of default goals from a project with a packaging of war. Running *mvn deploy* will run all the phases in the default lifecycle, each of which is associated with a plugin and a goal. Running *mvn compile exec:java* will run the compile phase followed by the java goal of the exec plugin.

Default Goals for JAR Packaging

Lifecycle Phase	Plugin:Goal	
compile	compiler:compile	
test-compile	compiler:testCompile	
test	surefire:test	
package	jar:jar	
install	install:install	to local repository
deploy	deploy:deploy	to remote repository

POM

To modify the default behaviour and add dependencies, configure the POM. The Super POM defines standard configuration of all projects. The effective POM is a merge of the POM and the super POM. To view the effective POM, right click pom.xml > Maven

```
<build>
    <plugins>
        <!--enable java 8-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.6.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
        <!--run exec:java goal to execute Java programs in the same VM-->
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.2.1</version>
            <executions>
                <execution>
                    <goals>
                        <goal>java</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <mainClass>console.Class1</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>

<reporting>
    <plugins>
        <!--creates html version of test results-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>2.19.1</version>
        </plugin>
        <!--generate javadoc documentation-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>2.10.4</version>
        </plugin>
    </plugins>
</reporting>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Jackson

```
package examples.featurespace;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.MapperFeature;
import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.IOException;
import java.util.GregorianCalendar;

public class JacksonTester {
    public static void main(String args[]) {
        try {
            //Serialize to JSON
            Film film1 = new Film("There's something about Mary", 2,
                 new GregorianCalendar(1997, 4, 27).getTime(), Genre.COMEDY);
            ObjectMapper mapper1 = new ObjectMapper();
            String jsonString = mapper1.writeValueAsString(film1);
            System.out.printf("Serialized object: %s%n",jsonString);

            //Deserialize from JSON
            String json = "{\"Id\":1,\"Title\":\"The Godfather\",
                          \"Released\":\"1972-01-01\",\"Stock\":4,\"Genre\":\"1\"}";
            ObjectMapper mapper2 = new ObjectMapper();
            mapper2.configure(
                          MapperFeature.ACCEPT_CASE_INSENSITIVE_PROPERTIES, true);
            Film film2 = mapper2.readerFor(Film.class).readValue(json);
            System.out.printf("Deserialized object: %s",film2);
        } catch (JsonParseException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.8.8</version>
</dependency>
```

Hamcrest

Hamcrest intends to make tests as readable as possible. To use Hamcrest matchers in JUnit you use the assertThat statement followed by one or several matchers.

```
public class FilmTest {

   @Test
   public void constructorShouldInitialiseFields() {
      // arrange and act
      Film film = new Film("The Pink Panther", 5, LocalDate.of(1964, 1, 20),
                                                                  Genre.COMEDY);
      Long id = film.getId();
      String title = film.getTitle();
      int stock = film.getStock();
      LocalDate released = film.getReleased();
      Genre genre = film.getGenre();

      // assert
      assertNull(id);
      assertThat(id,nullValue());

      assertEquals("The Pink Panther", title);
      assertThat(title,is(equalTo("The Pink Panther")));
```

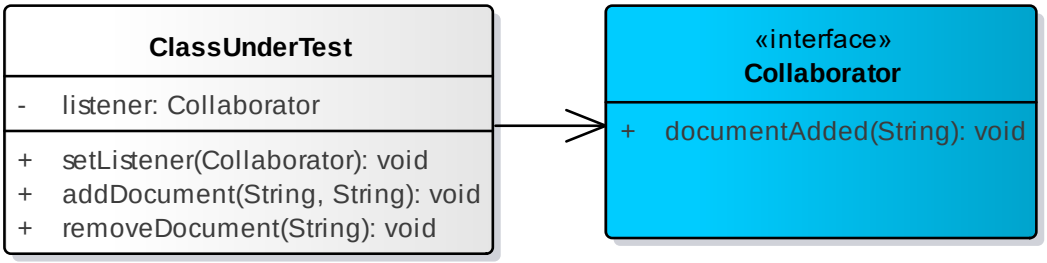
```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-library</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>
```

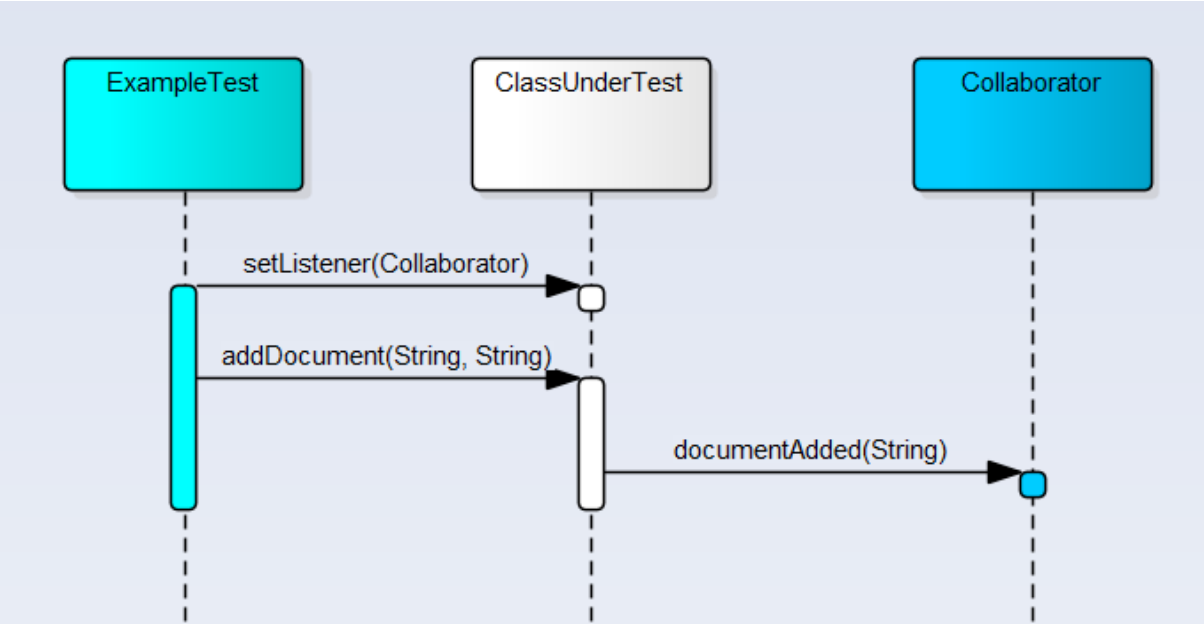
EasyMock

http://easymock.org/user-guide.html

```
<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>3.4</version>
    <scope>test</scope>
</dependency>
```

1. Create the mock
2. Have it set to the tested class
3. Record what we expect the mock to do
4. Tell all mocks we are now doing the actual testing
5. Test
6. Make sure everything that was supposed to be called was called





ExampleTest

```
package featurespace.easymock;

public class ExampleTest extends EasyMockSupport {
    private ClassUnderTest classUnderTest;
    private Collaborator collaborator;

    @Before
    public void setUp() {
        //1 Create the mock
        collaborator = mock(Collaborator.class);

        //2 Pass the mock into the class under test
        classUnderTest = new ClassUnderTest();
        classUnderTest.setListener(collaborator);
    }

    @Test
    public void addDocument() {
        // 3 Record what we expect the collaborator to do
        collaborator.documentAdded("New Document");
        // 4 Tell all mocks we are now doing the actual testing
        replayAll();
        // 5 Test
        classUnderTest.addDocument("New Document", "content");
        // 6 Make sure everything that was supposed to be called was called
        verifyAll();
    }
}
```

ExampleTest with annotations

```
public class ExampleTestWithAnnotations extends EasyMockSupport {

    @Rule
    public EasyMockRule rule = new EasyMockRule(this);
    //Mock injection on fields, so no need for a setter

    @Mock
    private Collaborator collaborator; // 1 Create the mock

    //2 Pass the mock into the class under test
    //Annotation is set on a field so that EasyMockRule will inject mocks created
    //with @Mock on its fields.
    @TestSubject
    private ClassUnderTest classUnderTest = new ClassUnderTest();

    @Test
    public void addDocument() {
        // 3 Record what we expect the collaborator to do
        collaborator.documentAdded("New Document");
        // 4 Tell all mocks we are now doing the actual testing
        replayAll();
        // 5 Test
        classUnderTest.addDocument("New Document", "content");
        // 6 Make sure everything that was supposed to be called was called
        verifyAll();
    }
}
```

InMemoryFilmDAOTest

```
package featurespace.easymock;

public class InMemoryFilmDAOTest extends EasyMockSupport {

    private InMemoryFilmDAO classUnderTest;
    private Serializer serializer;
    private Map<Long, Film> films;

    @Before
    public void setUp() {
        // 1 Create the mocks
        serializer = mock(Serializer.class);
        //niceMock avoids AssertionError for unexpected method calls
        films = niceMock(HashMap.class);
        //pass mock objects into class under test
        classUnderTest = new InMemoryFilmDAO(films, serializer);
    }

    @Test
    public void selectAllShouldCallDeserializeMethodOfSerializer() {
        // 3 Record what we expect the mock to do
        expect(serializer.deserialize()).andReturn(new HashMap<>());
        // 4 Tell all mocks we are now doing the actual testing
        replayAll();
        // 5 Test
        classUnderTest.selectAll();
        // 6 Make sure everything that was supposed to be called was called
        verifyAll();
    }

    @Test
    public void insertShouldCallSerializeMethodOfSerializer() {
        // 3 Record what we expect the mock to do
        serializer.serialize(new HashMap<>());
        // 3 Record what we expect the mock to do
        expect(films.values()).andReturn(new ArrayList<Film>());
        // 4 Tell all mocks we are now doing the actual testing
        replayAll();
        // 5 Test
        classUnderTest.insert(new Film());
        // 6 Make sure everything that was supposed to be called was called
        verifyAll();
    }
}
```

InMemoryFilmDAOTest with Annotations

```
public class InMemoryFilmDAOTestWithAnnotations extends EasyMockSupport {

   @Rule
   public EasyMockRule rule = new EasyMockRule(this);
   //Mock injection on fields of InMemoryFilmDAO

   @Mock  private Serializer serializer; // 1 Create the mocks
   @Mock(type = MockType.NICE) private Map<Long, Film> films; // ignore unexpected
                                                              // method call

  //TestSubject annotation set on a field so that EasyMockRule will inject mocks
  //created with Mock on its fields.
  @TestSubject
  private InMemoryFilmDAO classUnderTest = new InMemoryFilmDAO();

  //Test methods as above
```

Style guide

https://google.github.io/styleguide/javaguide.html

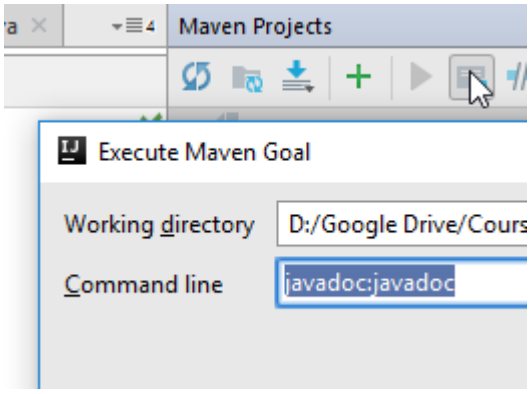
Javadoc

https://maven.apache.org/plugins/maven-javadoc-plugin/

Add the plugin to the repoting section of the POM

```
<reporting>
    <plugins>
        <!--creates html version of test results-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-report-plugin</artifactId>
            <version>2.19.1</version>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>2.10.4</version>
        </plugin>

    </plugins>
</reporting>
```

Click the Execute Maven Goal button and run the javadoc goal of the javadoc plugin (javadoc:javadoc)



Immutable types

Immutable objects have many advantages, including:

- Safe for use by untrusted libraries.
- Thread-safe
- All immutable collection implementations are more memory-efficient than their mutable siblings
- Can be used as a constant, with the expectation that it will remain fixed

Immutable classes have

- No mutators
- Private fields
- Final class prevents overriding methods

An immutable class

```
package java.time;
public final class LocalDate {
    private final int year;
    private final short month;
    private final short day;

    //returns a copy of this LocalDate with the specified number of days added
    public LocalDate plusDays(long daysToAdd) {
        if (daysToAdd == 0) {
            return this;
        }
        long mjDay = Math.addExact(toEpochDay(), daysToAdd);
        return LocalDate.ofEpochDay(mjDay);
    }
```

A mutable class

```
package com.sun.javafx.geom;
public class Point2D {
    public float x;
    public float y;
    public void setLocation(float x, float y) {
        this.x = x;
        this.y = y;
    }
```

Guava

Guava is a set of core libraries that includes immutable collections, a graph library, functional types, an in-memory cache, and APIs/utilities for concurrency, I/O, hashing, primitives, reflection and string processing.

https://github.com/google/guava/wiki

Immutable collections

When you don't expect to modify a collection, or expect a collection to remain constant, it's a good practice to defensively copy it into an immutable collection.

```
public static final ImmutableSet<String> COLOR_NAMES = ImmutableSet.of(
        "red",
        "orange",
        "yellow",
        "green",
        "blue",
        "purple");
```

Apache Commons

https://commons.apache.org/components.html

Math

```
long a = NumberUtils.toLong("5");//0 if conversion fails
```

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.5</version>
</dependency>
```

CSV

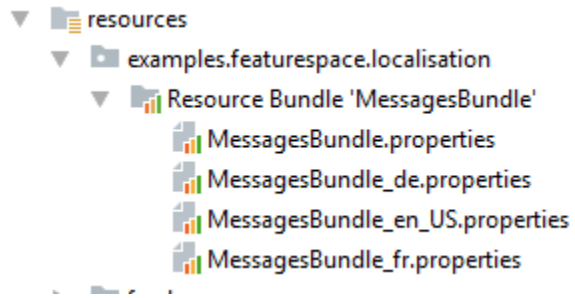
```
Reader in = new FileReader("…apache/book1.csv");
Iterable<CSVRecord> records =
                CSVFormat.DEFAULT.withHeader("Last Name", "First Name").parse(in);
for (CSVRecord record : records) {
    String lastName = record.get("Last Name");
    String firstName = record.get("First Name");
    System.out.printf("%s %s%n",firstName,lastName);
}
```

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-csv</artifactId>
    <version>1.4</version>
</dependency>
```

Localisation

```
package examples.featurespace.localisation;

import java.util.Locale;
import java.util.ResourceBundle;

public class I18NSample {
    static public void main(String[] args) {
        Locale.setDefault(new Locale("de", "fr"));
        ResourceBundle messages = ResourceBundle.getBundle(
                          "examples.featurespace.localisation.MessagesBundle");
        System.out.println(messages.getString("greetings"));
        System.out.println(messages.getString("inquiry"));
        System.out.println(messages.getString("farewell"));
    }
}
```

Add properties files to the resources folder



Sample exam questions

Exam 1 syllabus

Exam 2 syllabus