Java Programming Course

Contents

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
  - File > Settings > Editor > General > Show quick documentation on mouse move
  - File > Project structure > Project settings > Project language level
  - View > Tool Buttons
  - Right click Gradle tab > Move to > left
  - View > Tool Windows > TODO
- 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

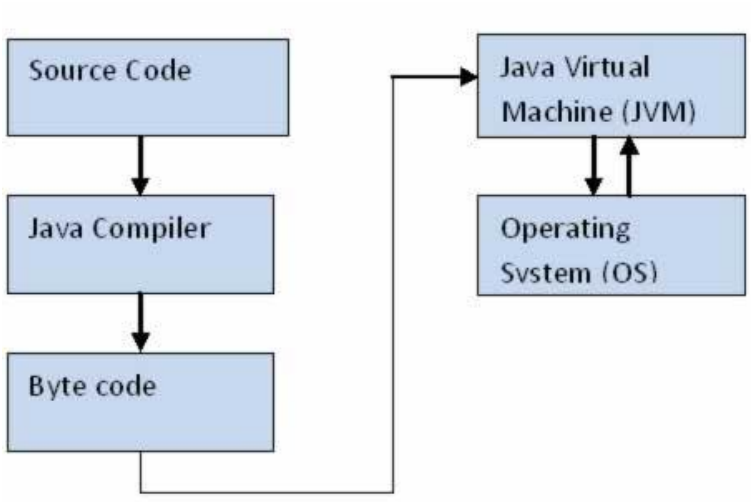
```
File > Project Structure > Platform Settings > SDK
JavaSE:                 http://docs.oracle.com/javase/8/docs/api
JavaFX:                 https://docs.oracle.com/javase/8/javafx/api

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/
```

- eclipse

```
eclipse { classpath {  downloadJavadoc = true    }  }
```

Fundamentals

Java Architecture



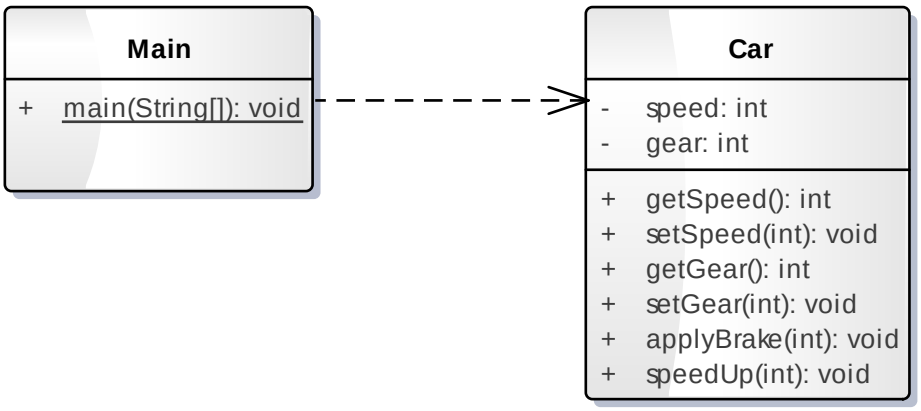
The Java Runtime Environment (JRE) is part of the Java Development Kit, a set of programming tools for developing Java applications. The JRE provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

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.

UML Class Diagrams




```
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());
	}
}

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

Access level 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

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);
}
```

Factorial calculator

```
public class Main {
	public static void main(String[] args) {
		int result = Maths.factorial(6); //6 x 5 x 4 x 3 x 2
		System.out.println(result);
	}
}


public class Maths {
	public static int factorial(int i) {
		return 0;
	}
}
```

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

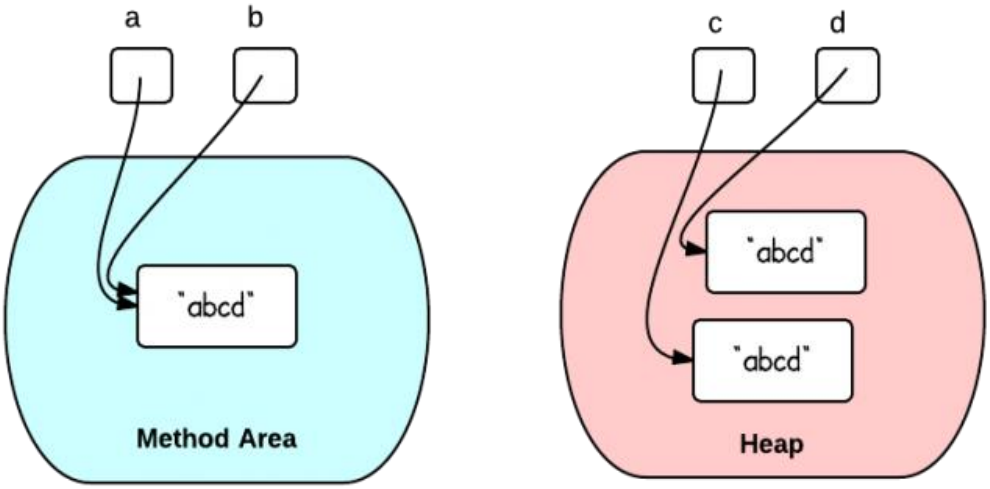
```
String a = "ab";
a = a + "cd"; //new object is created
```

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.

Enum Types

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

```
DayOfWeek day = DayOfWeek.MONDAY;

if(day == DayOfWeek.TUESDAY){
}
```

Enums have methods inherited from the Enum class, including `values`, which returns an array containing the values of the enum

```
DayOfWeek[] days = DayOfWeek.values();

for (DayOfWeek dayOfWeek : days) {
	System.out.println(dayOfWeek);
}
```

An enum is defined as follows

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

Using a switch block with an enum

```
DayOfWeek day = DayOfWeek.MONDAY;

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

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

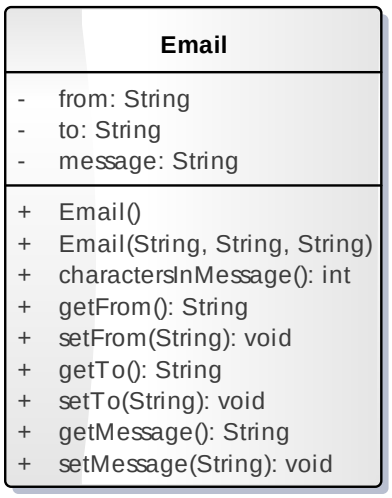
```
Car car1 = new Car(50,4);
```

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

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

Email class



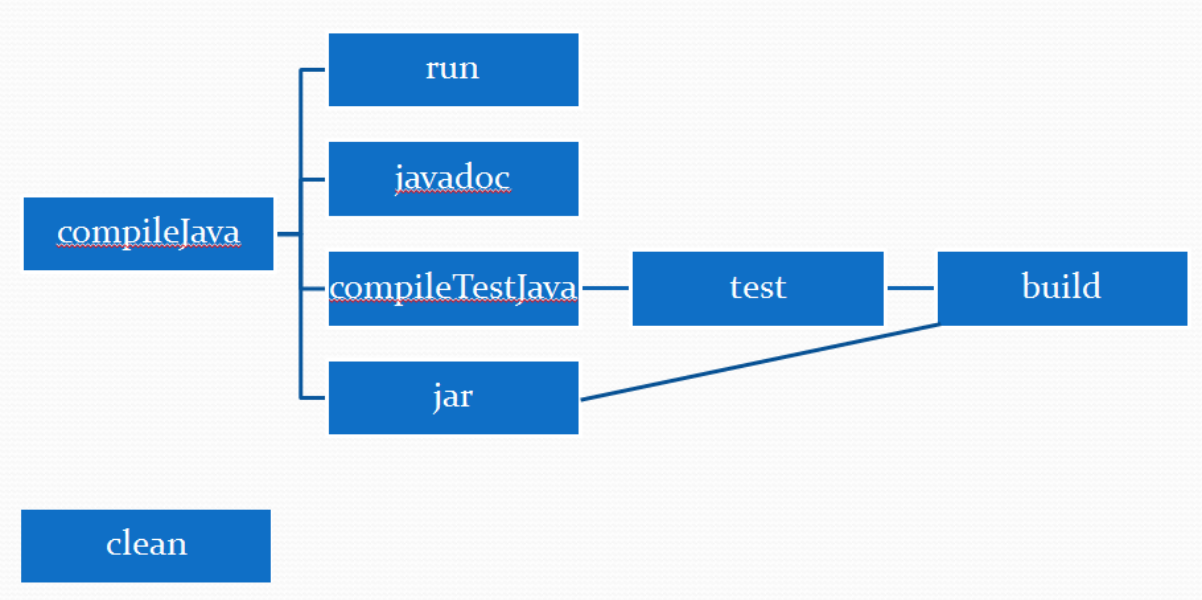
This UML diagram describes an email in terms of state: the from, to and message fields and behaviour: the get and set methods and a charactersInMessage method that returns a value that can be calculated by calling the length() method of the String class. There are also two constructors.

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'
apply plugin: 'eclipse'

mainClassName = "com.example.Class1"

sourceCompatibility = 1.8

version = '1.0'

repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
eclipse {
    classpath {
        downloadJavadoc = true
    }
}
```

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

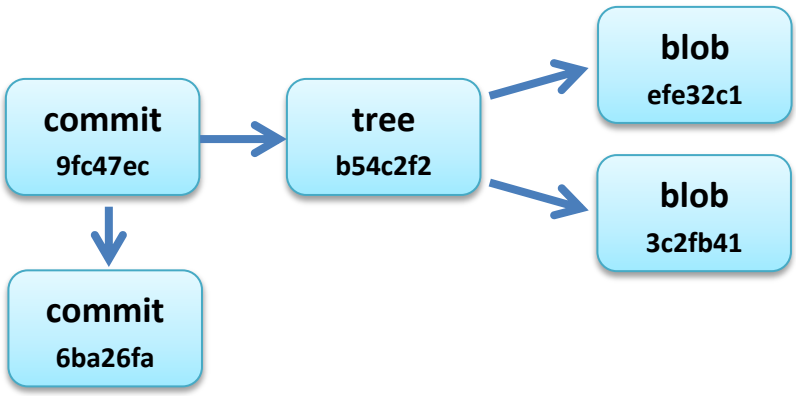
Git is a distributed revision control system with an emphasis on speed, data integrity and support for distributed, non-linear workflows.

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

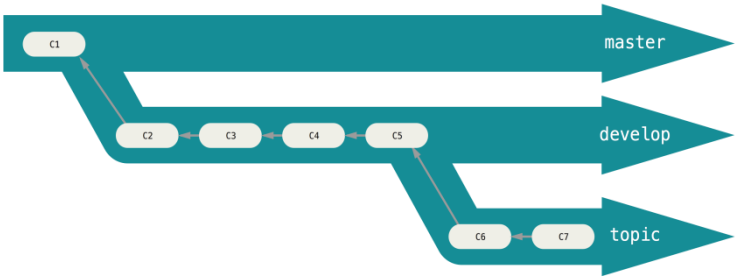
3. commit files in the index; this creates a snapshot of the project
   i. tree object with checksum is generated for project directories
   ii. commit object with checksum includes metadata and points at root directory and previous commit
   iii. >git commit –m “message”



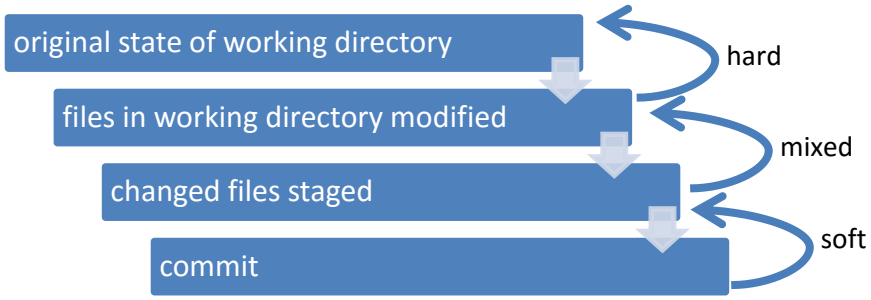
Branching

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

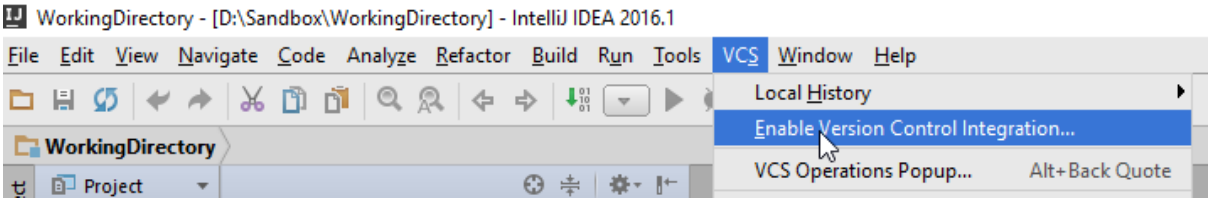


git reset [<mode>] [<commit>] resets the current branch head to <commit> and possibly updates the index (resetting it to the tree of <commit>) and the working tree depending on <mode>. The mode can be soft, mixed or hard.



Create a local and repository

1. Create a local repository in IntelliJ



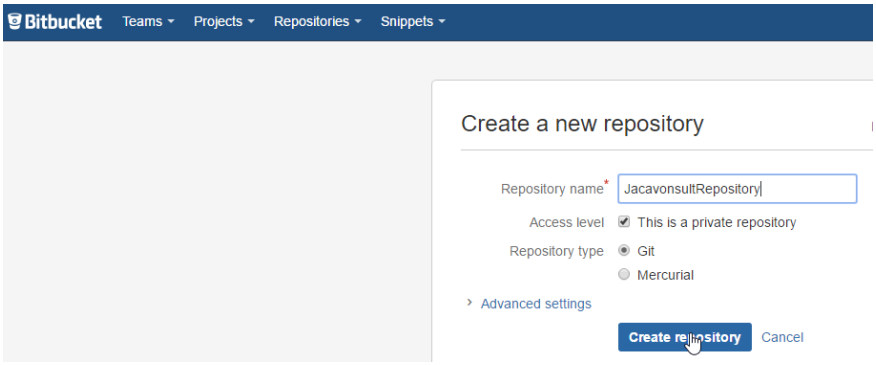
2. Add a .gitignore file (see https://www.gitignore.io/)

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

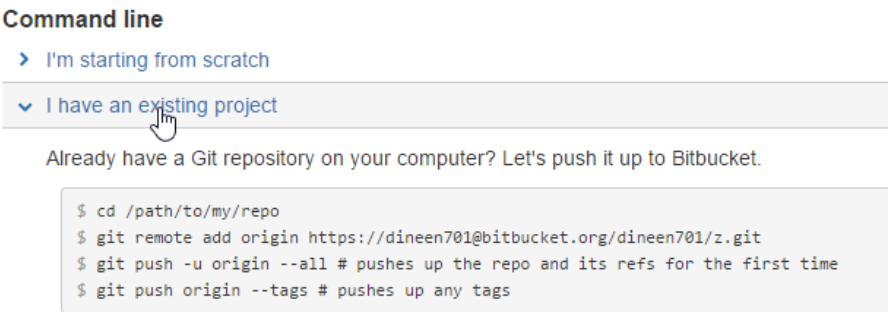
3. Open the working directory in source tree
4. Commit in master branch
5. Add develop branch

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



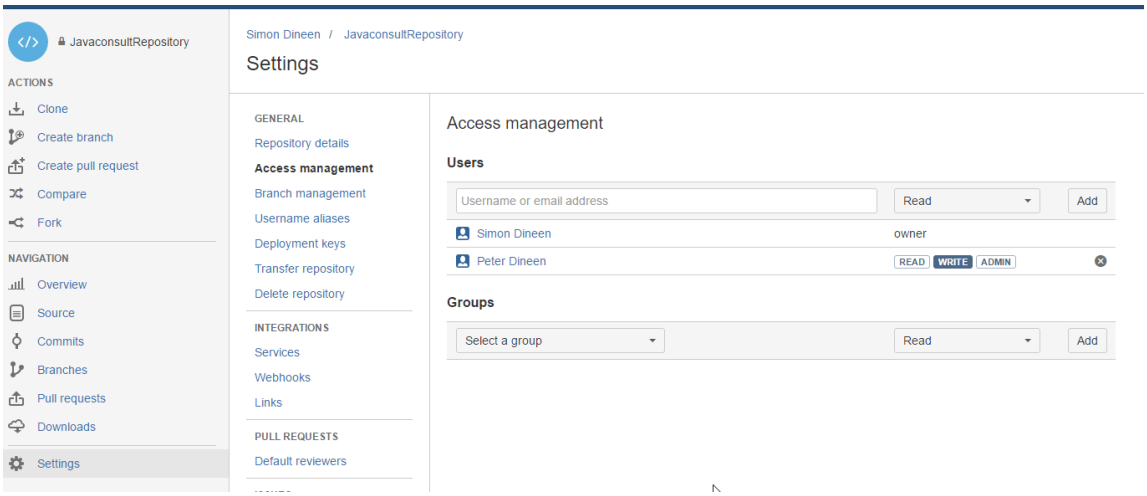
3. To add a new remote, open the source tree console and use the git remote add command. This takes two arguments; a remote name, for example, origin, and a remote URL, for example, https://dineen701@bitbucket.org/dineen701/java-course.git

```
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

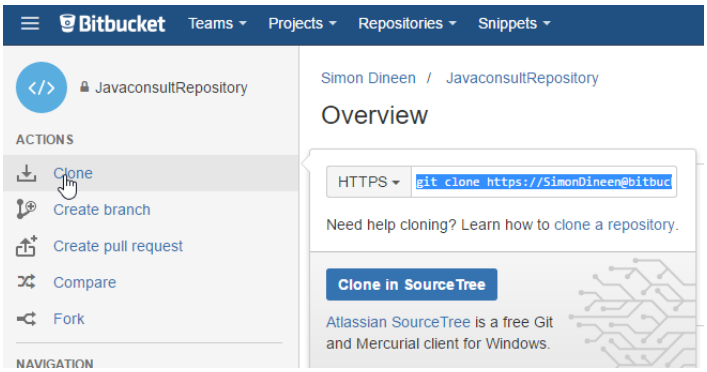
a. select repository in Bitbucket
b. click Settings button on left
c. select Access management



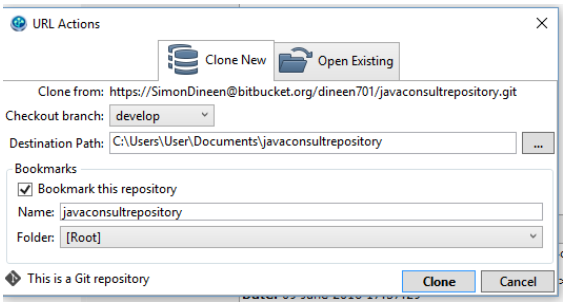
5. type username, select Write access

Cloning the repository

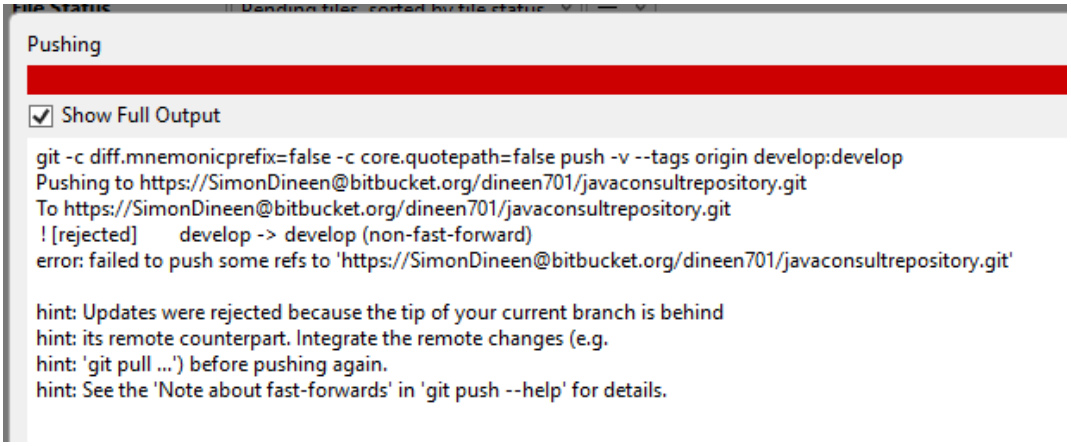
6. Another user logs in to bitbucket
7. Click “Clone in sourcetree”



8. Checkout develop branch

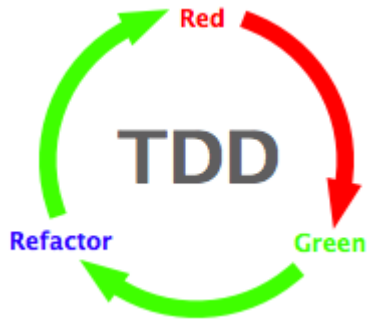


9. Open project in intellij (see destination path above)
10. Edit the working directory
11. Commit and push
12. The other developer pulls the branch from source tree
13. If a developer pushes a commit to the remote repository, another developer attempting to push before pulling will encounter an error



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.

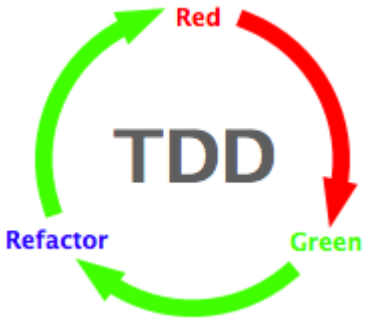
Phases when writing a test

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



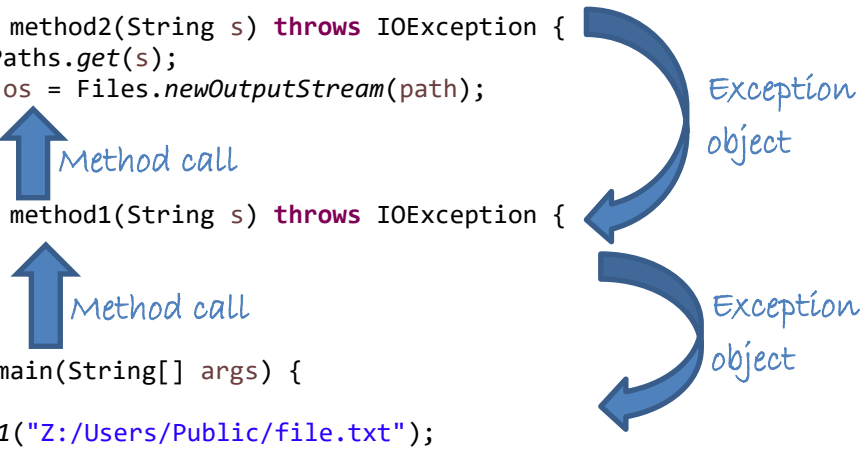
Apply this to the above example

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.

Exceptions

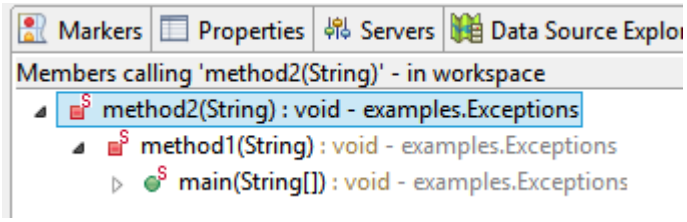
An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
A method can either handle an exception, or alternatively throw the exception object down the method call stack.

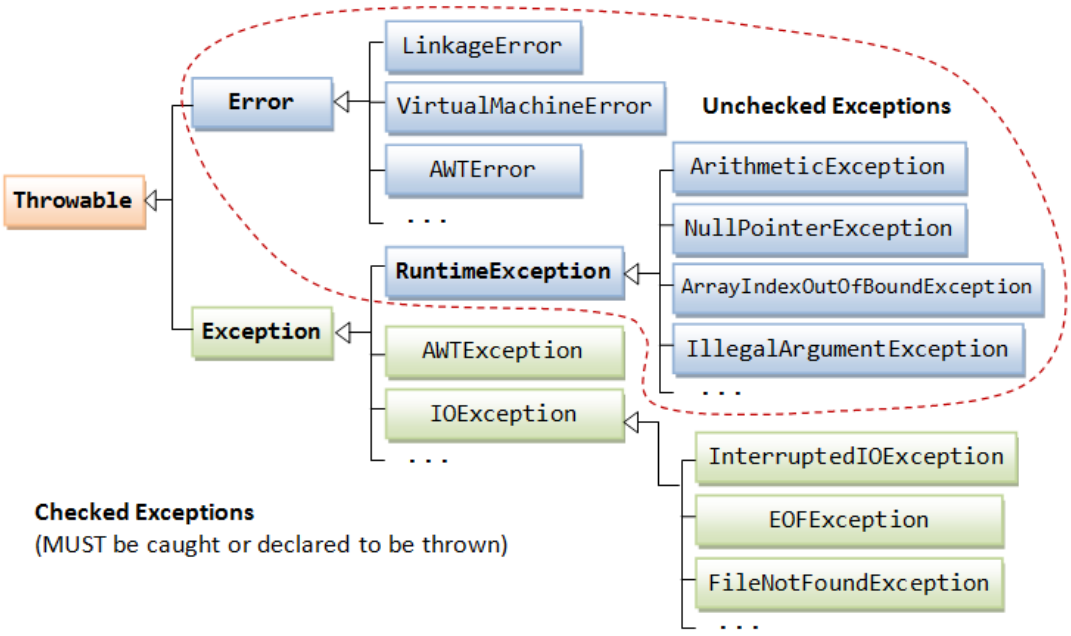
```
public class Main {

	private static void method2(String s) throws IOException {
		Path path = Paths.get(s);
		OutputStream os = Files.newOutputStream(path);
	}

	private static void method1(String s) throws IOException {
		method2(s);
	}

	public static void main(String[] args) {
		try {
			method1("Z:/Users/Public/file.txt");
		} catch (IOException e) {
			System.out.println(e);
		}
	}
}
```



Paths.get can throw an InvalidPathException, which is a RuntimeException, so handling isn't enforced. However, Files.newOutputStream can throw an IOException; since this isn't a RuntimeException, the compiler ensures that the exception is either declared with throws keyword, or handled with a try-catch block.

To view the method call stack, right click method2 and select "open call hierarchy"





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.

Build a derived class of RuntimeException named FilmException

Multiple catch and finally blocks

A catch block can be followed with multiple catch blocks and an optional finally block, which is always executed

```
private static void catchAndFinally() throws IOException {
	OutputStream os = null;
	try {
		Path path = Paths.get("C:/file.txt");
		os = Files.newOutputStream(path);
	} catch (AccessDeniedException e) {
		System.out.println("first catch block " + e);
	} catch (IOException e) {
		System.out.println("second catch block " + e);
	} finally {
		System.out.println("finally block");
		if (os != null)
			os.close();
	}
}
```

The AutoCloseable interface

The close() method of an AutoCloseable object is called automatically when exiting a try-with-resources block

```
	private static void catchAndFinally3() throws IOException {
		Path path = Paths.get("C:/file.txt");
		try (OutputStream os = Files.newOutputStream(path)) {

		} catch (IOException e) {
			System.out.println("catch block " + e);
		}
	}
```

Throwing an exception

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

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

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);
}
```

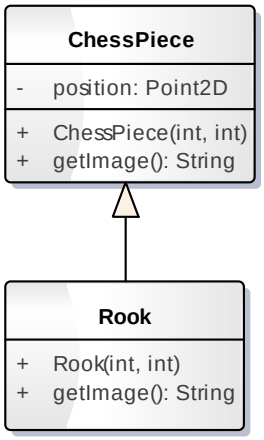
Following the TDD rhythm

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

OO Basics

Inheritance and overriding

A subclass inherits all non-private members (fields, methods, and nested classes) from its superclass. Constructors aren't members, so they're not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

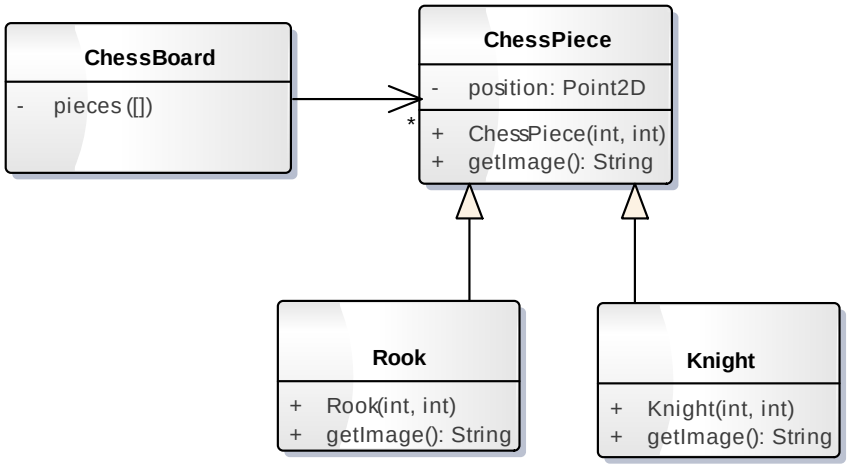



```
package chess;
import javafx.geometry.Point2D;

class ChessPiece {
	private Point2D position;
	public ChessPiece(int x, int y) {
		position = new Point2D(x,y);
	}
	public String getImage() {
		return null;
	}
}

class Rook extends ChessPiece {
	public Rook(int x, int y) {
		super(x, y);
	}
	@Override
	public String getImage() {
		return super.getImage();
	}
}
```

The first expression in a derived class constructor is an implicit call to the no-argument constructor in the base class. To call a different constructor, use the super keyword.
By overriding the getImage method, subclasses of ChessPiece could describe how they're displayed.

Composition

Another way of associating classes is to use composition. A ChessBoard is composed of multiple ChessPiece objects.



Polymorphism

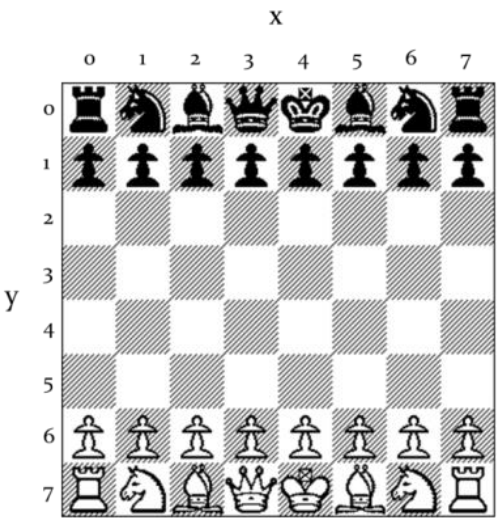
Polymorphism in biology describes the occurrence of more than one form or morph

Light-morph jaguar

Dark-morph jaguar

Applied to programming, subclasses can define their own unique behaviours and yet share some of the same functionality of the parent class. For example, iterating through an array of ChessPiece objects and calling the getImage method of each object, the method of the currently referenced subclass object will be called.

```
package chess;
import javafx.geometry.Point2D;

class Main {
	public static void main(String[] args) {
		ChessBoard board = new ChessBoard();
		for (ChessPiece cp : board.pieces) {
			String image = cp.getImage();
		}
	}
}

class ChessPiece {
	private Point2D position;
	public ChessPiece(int x, int y) {
		position = new Point2D(x,y);
	}
	public String getImage() {
		return null;
	}
}

class Rook extends ChessPiece {
	public Rook(int x, int y) {
		super(x, y);
	}
	@Override
	public String getImage() {
		return "rdl";//rook dark on light square
	}
}

class Knight extends ChessPiece {
	public Knight(int x, int y) {
		super(x, y);
	}
	@Override
	public String getImage() {
		return "nld";//knight light on dark square
	}
}


class ChessBoard {
	public ChessPiece[] pieces =
			new ChessPiece[32];

	public ChessBoard() {
		pieces[0] = new Rook(0,0);
		pieces[1] = new Knight(1,0);
		pieces[2] = new Bishop(2,0);
	}

}
```



Abstract classes

Abstract classes can't be instantiated. They can contain abstract methods whose implementation is deferred to a derived class, such as Rook in the following example. The compiler enforces polymorphism by ensuring that all classes that derive from ChessPiece override the abstract method.

```
abstract class ChessPiece {

	private Point2D position;

	public ChessPiece(int x, int y) {
		position = new Point2D(x,y);
	}

	public abstract String getImage();
}

class Rook extends ChessPiece {

	public Rook(int x, int y) {
		super(x, y);
	}

	@Override
	public String getImage() {
		return null;
	}
}
```

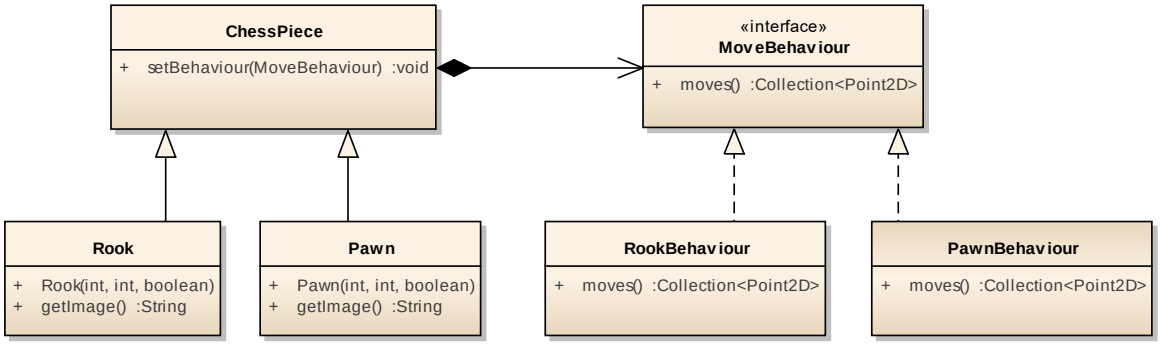
Strategy

General design principles include

1. encapsulate what varies
2. favour composition over inheritance
3. program to interfaces not implementations

Design patterns describe templates for solving commonly occurring problems. The strategy pattern follows these principles by defining a family of algorithms; encapsulating each one and making them interchangeable.

Encapsulation means hiding the properties and behaviours of an object and allowing outside access only as appropriate.



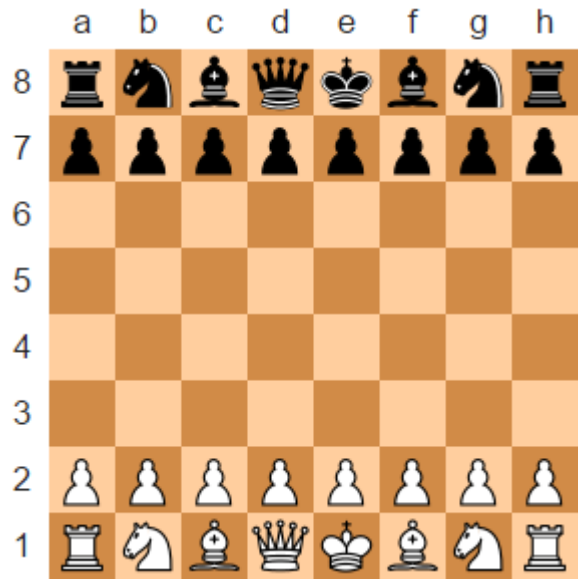
Interfaces

An interface is a specification describing the methods of an object. The implementation of these methods is deferred to a class. Interfaces are similar to abstract classes; a distinction is that a class can implement multiple interfaces but can only derive from one immediate base class.

```
public interface MoveBehaviour {
	Collection<Point2D> moves(Point currentSquare);
}

public class RookBehaviour implements MoveBehaviour {
	@Override
	public Collection<Point2D> moves(Point currentSquare) {
		// TODO Auto-generated method stub
		return null;
	}
}

 class JavaFX {
	public static void main(String[] args) {
		ChessPiece cp = new Rook(0, 1);
		cp.setBehaviour(new RookBehaviour());
```

Repair test failure

The Knight at b8 is rendered incorrectly



Run the test phase
>gradle test
There's a build failure. See build/reports/tests/index.html
Repair the getImage method in the Knight class so that the assertion passes.
The image filenames comprise three characters;

1. the first indicates the type of piece (n for Knight)
2. the second is the colour of the piece; d for dark and l for light
3. the third is the colour of the square

Parameterised tests

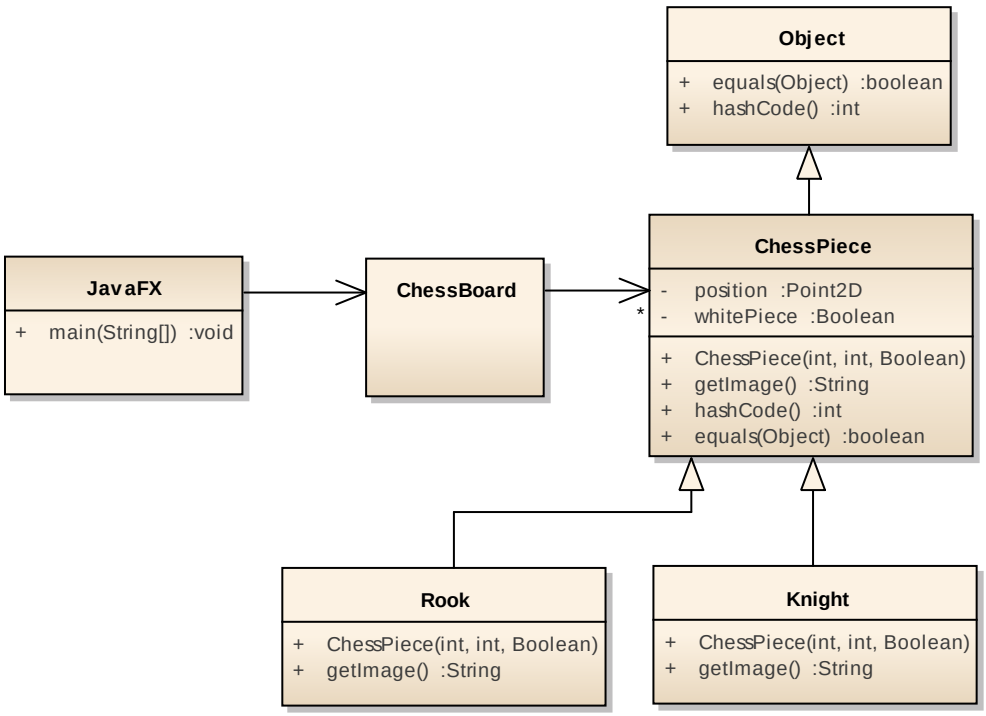
Parameterized tests enable a test to be run repeatedly with different values. This requires the JUnitParams dependency to be added to the POM. Values are passed into the @Parameters annotation as a String array. Each element in the array is a comma-separated set of values which must match the method parameters in order and type.

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.+'
    testCompile 'pl.pragmatists:JUnitParams:1.0.4'
```

```
package chess;

import static org.junit.Assert.*;
import junitparams.JUnitParamsRunner;
import junitparams.Parameters;

import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(JUnitParamsRunner.class)
public class ParameterizedTest {

	@Test(expected=IndexOutOfBoundsException.class)
	@Parameters({"0,8", "8,0", "-1,0", "0,-1"})
	public void chessPieceConstructorThrowsExceptionIfNotOnBoard(int x, int y)
	{
		new Rook(x,y,true);
	}

	@Test
	@Parameters({"0,0,false,a8","7,0,false,h8","0,7,true,a1","7,7,true,h1"})
	public void shouldConvertToAlgebraicNotation(int x, int y, boolean isWhite,
                                                 String expectedAlgebraic) {
		//arrange
		ChessPiece cp = new ChessPiece(x,y,isWhite);

		//act
		String actualAlgebraic = cp.getAlgebraicNotatation();

		//assert
		assertEquals(expectedAlgebraic, actualAlgebraic);
	}
}
```

The Object class

Every class is a direct or indirect descendant the object class, whose methods can be overridden.



The equals() method provided in the Object class tests whether the object references are equal, which will be true if the references are pointing at the same object. To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the equals()method.

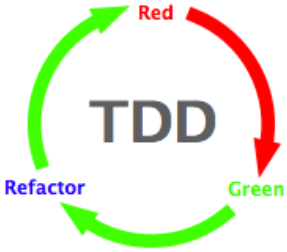
```
package chess;
import static org.junit.Assert.*;
import org.junit.Test;
public class OverrideObjectTest {
	@Test
	public void referenceEquality() {
		//arrange
		Rook rook1 = new Rook(1, 2, true);
		Rook rook2 = rook1;
		//assert
		assertTrue(rook1.equals(rook2));
	}
	@Test
	public void valueEquality() {
		//arrange
		Rook rook1 = new Rook(1, 2, true);
		Rook rook2 = new Rook(1, 2, true);
		//assert
		assertTrue(rook1.equals(rook2));//only true if equals overridden
	}
```

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 equalHashcodes() {
	//arrange
	Rook rook1 = new Rook(1, 2, true);
	Rook rook2 = new Rook(1, 2, true);

	//act
	int i = rook1.hashCode();
	int j = rook2.hashCode();

	//assert
	assertTrue(i==j);//only true if hashCode overridden
}
```

Override equals and hashCode methods in the Film class

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



1. run the test; it should fail
2. override the methods in the object class so that the assertions pass
3. refactor

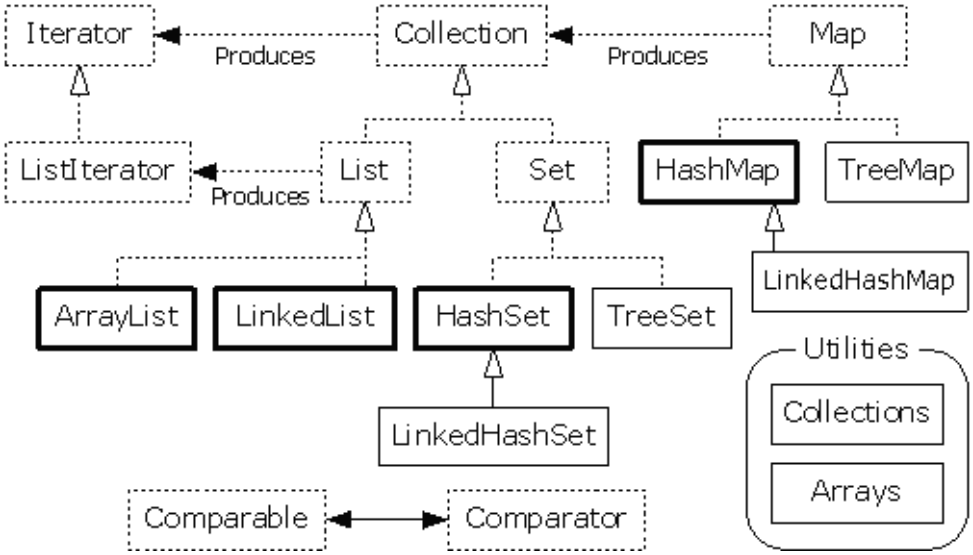
```
@Test
public void filmsWithSameTitleShouldBeEqual () {
	//arrange
	Film film1 = new Film();
	film1.setTitle("The Godfather");
	Film film2 = new Film();
	film2.setTitle("The Godfather");

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

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

Overview



Interfaces in the collection hierarchy

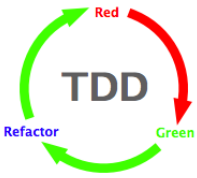
- Collection is the root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered
- A List is an ordered collection. The user has precise control over where 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.

The java.util package contains the collections framework

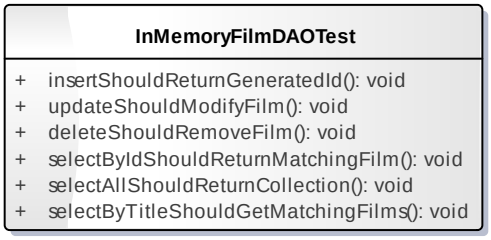
Using collections classes

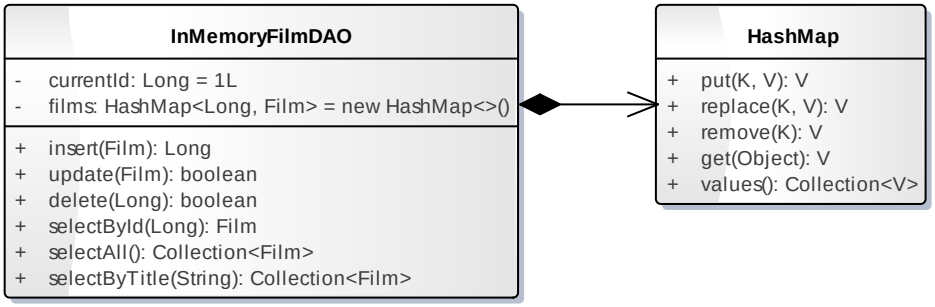
```
public class CollectionTest {

	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.

The solid diamond symbol is a composition,

a type of association that indicates ownership.

Lambda Expressions

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

Functional interfaces have one abstract method; they encapsulate a block of code. For example, the Runnable interface encapsulates code that can be executed in a separate thread. There are three ways of creating an instance of this interface:

1. A named class

```
class X implements Runnable {
	@Override
	public void run() {
		// executed in separate thread
	}
}

Runnable r = new X();
```

2. An anonymous class

```
Runnable r = new Runnable() {
	@Override
	public void run() {
		// executed in separate thread
	}
};
```

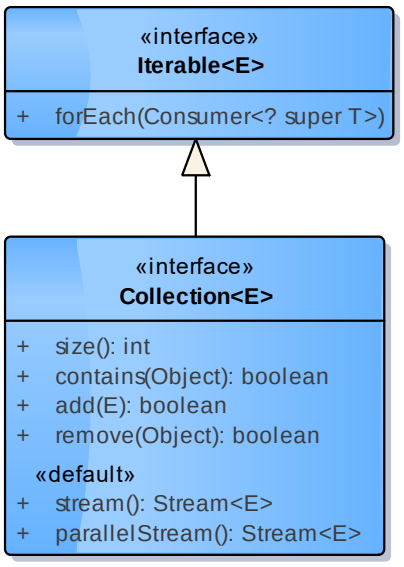
3. A lambda expression

```
Runnable r = () -> {
	// executed in separate thread
};
```

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
    }
}
```



The Stream API

A stream is an abstraction of a sequence that can be filtered, transformed and ordered.

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();

Predicate<String> predicate = s->s.contains("Europe");

Function<String, String> function = s -> s.substring(7);

long count =
   zoneIds.stream().     //create a stream
   filter(predicate).    //intermediate operation
   map(function).        //intermediate operation
   count();              //terminal operation
```

Predicate<T> is a functional interface with an abstract method that takes a generic argument of type T and returns a Boolean

Function<T,R> is a functional interface with an abstract method that takes a T argument and returns a R

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.

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();

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

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

//combiner - function for combining two values, compatible with the accumulator
BiConsumer<ArrayList<String>, ArrayList<String>> combiner =
                                                (x, y) -> { x.addAll(y); };


List<String> zones =
zoneIds.stream().                           //create a stream
filter(s -> s.contains("Europe")).          //intermediate operation
map(x -> x.substring(7)).                   //intermediate operation
collect(supplier, accumulator, combiner);   //terminal operation
```

The forEach default method of the Iterable interface can be used to print each element in the List. The method takes a consumer argument. A consumer represents an operation that accepts a single input argument and returns no result.

```
//action - The action to be performed for each element
Consumer<String> action = s -> System.out.println(s);
zones.forEach(action);
```

Rewriting this, passing lambda expressions into the methods

```
List<String> zones = zoneIds.stream().
      filter(s -> s.contains("Europe")).
      map(x -> x.substring(7)).
      collect(
            () -> new ArrayList<String>(), //supplier
            (x, y) -> {x.add(y);},         //accumulator
            (x, y) -> {  x.addAll(y);}     //combiner
      );


zones.forEach( s -> System.out.println(s));
```

Method References

Lambda expressions are used to create anonymous methods. In cases where an existing method is called, a method reference can be clearer. This shorthand notation enables a method of an object to be called for each element in a sequence.

```
Set<String> zoneIds = ZoneId.getAvailableZoneIds();

List<String> zones = zoneIds.stream().
       filter(s -> s.contains("Europe")).
       map(x -> x.substring(7)).
       collect(
              () -> new ArrayList<String>(), //ArrayList::new
              (x, y) -> {x.add(y);},         //ArrayList::add
              (x, y) -> {  x.addAll(y);}     //ArrayList::addAll
       );

zones.forEach(s -> System.out.println(s)); // System.out::println
```

putting these method references into the above code:

```
       List<String> zones = zoneIds.stream().
              filter(s -> s.contains("Europe")).
              map(x -> x.substring(7)).
              collect(
                     ArrayList::new,     //supplier
                     ArrayList::add,     //accumulator
                     ArrayList::addAll   //combiner
           );

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

An overload of the collect method takes a Collector argument

```
       List<String> zones = zoneIds.stream().
                     filter(s -> s.contains("Europe")).
                     map(x -> x.substring(7)).
                     collect(Collectors.toList());
```

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, tempusfugitlibrary.org, assists with running test methods in parallel. Each test method within a class will run on its own thread and in parallel with any other test methods in that class. So, the number of threads for a given test class will be equal to the number of test methods within that class.

The following example repeatedly calls the insert method of the shared InMemoryFilmDAO instance from two threads.

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import com.google.code.tempusfugit.concurrency.ConcurrentTestRunner;

@RunWith(ConcurrentTestRunner.class)
public class ParallelTest {
	private static InMemoryFilmDAO sut = new InMemoryFilmDAO();
	private Film film = new Film();
	private int n = 100;

	@Test
	public void shouldRunInParallel1() throws InterruptedException {
		for (int i = 0; i < n; i++) {
			//act
			Long id1 = sut.insert(film);
		}
		Thread.sleep(1000);//pause to allow threads to finish
		assertEquals(n*2, sut.selectAll().size());
	}

	@Test
	public void shouldRunInParallel2() {
		for (int i = 0; i < n; i++) {
			//act
			Long id1 = sut.insert(film);
		}
	}
}
```

This requires the following dependency

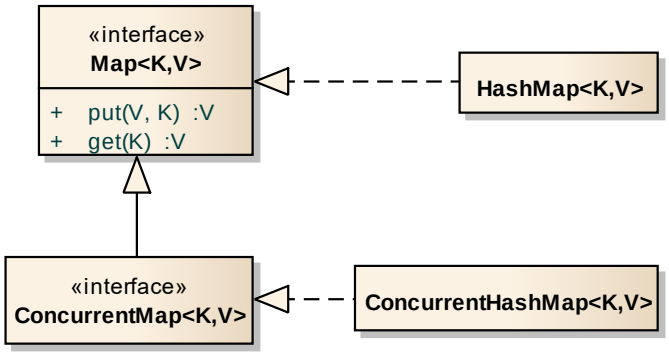
```
dependencies {
    testCompile 'com.google.code.tempus-fugit:tempus-fugit:1.1'
```

java.util.concurrent

AtomicLong

The java.util.concurrent.atomic 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 getAndIncrement() method of AtomicLong.

Concurrent collections



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:

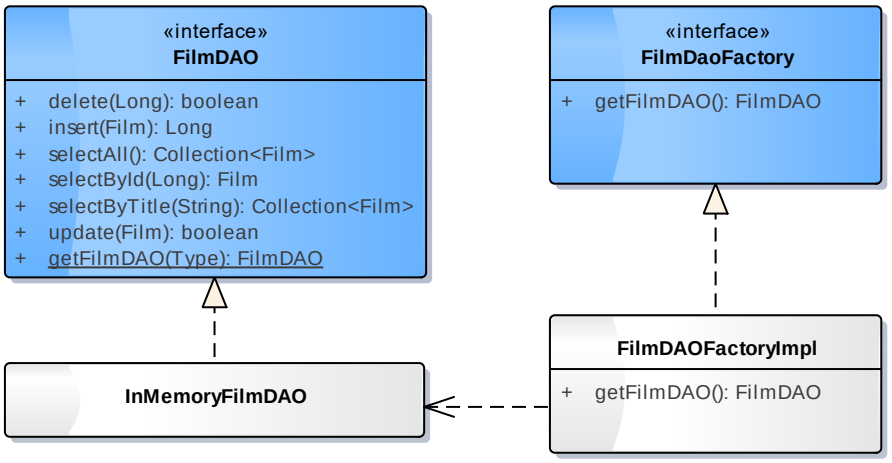
```
List list = Collections.synchronizedList(new ArrayList());
```

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.

Factory design pattern

The factory method pattern is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created. This is done by creating objects via calling a factory method specified in an interface and implemented by child classes rather than by calling a constructor.

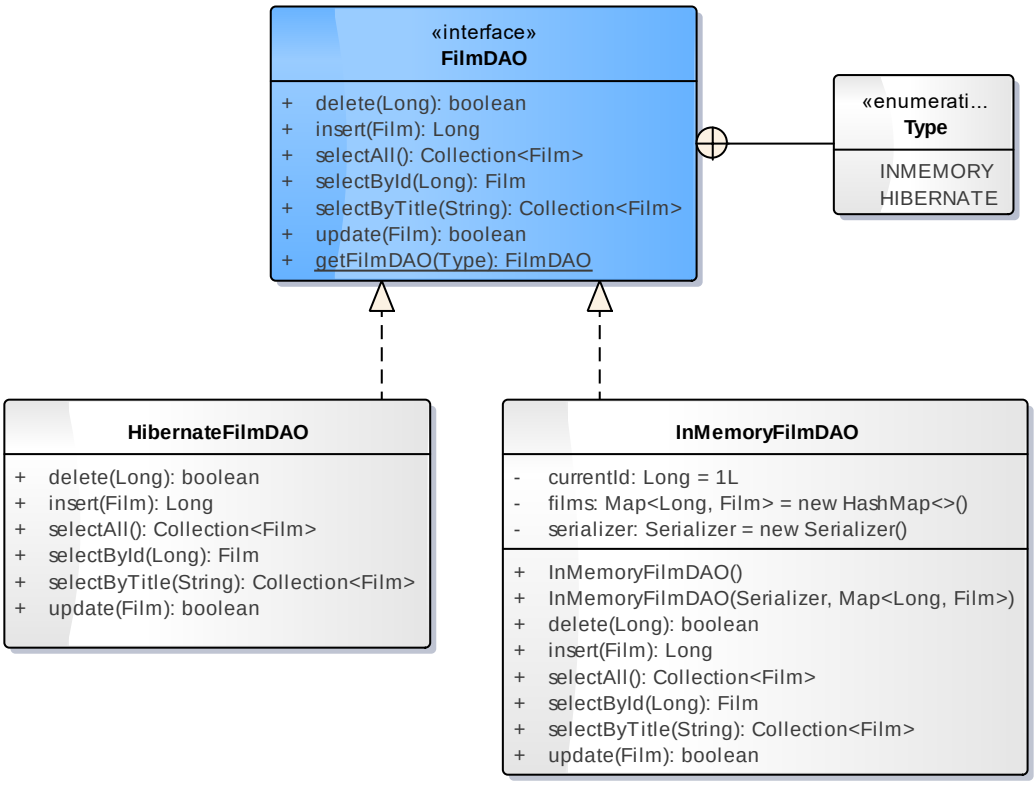


OO Design Principles

1. encapsulate what varies
2. favour composition over inheritance
3. program to interfaces not implementations
4. **depend on abstractions, not concrete classes**

A variation on this pattern would be to add a static factory method to the FilmDAO interface, enabling a reference to an InMemoryFilmDAO instance being obtained with the expression

```
FilmDAO dao = FilmDAO.getFilmDAO(Type.INMEMORY);
```



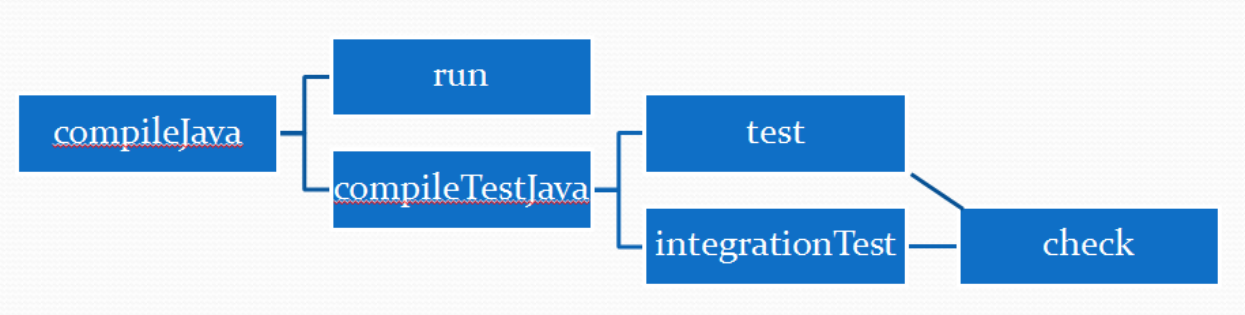
```
public interface FilmDAO {
	public enum Type {
		INMEMORY, JPA
	}
	boolean delete(Long filmId);
	Long insert(Film film);
	Collection<Film> selectAll();
	Film selectById(Long id);
	Collection<Film> selectByTitle(String search);
	boolean update(Film film);
	static FilmDAO getFilmDAO(Type type) {
		switch (type) {
		case INMEMORY:
			return new InMemoryFilmDAO();
		case JPA:
			return new JpaFilmDAO();
		default:
			return null;
		}
	}
}
```

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
```



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);
```

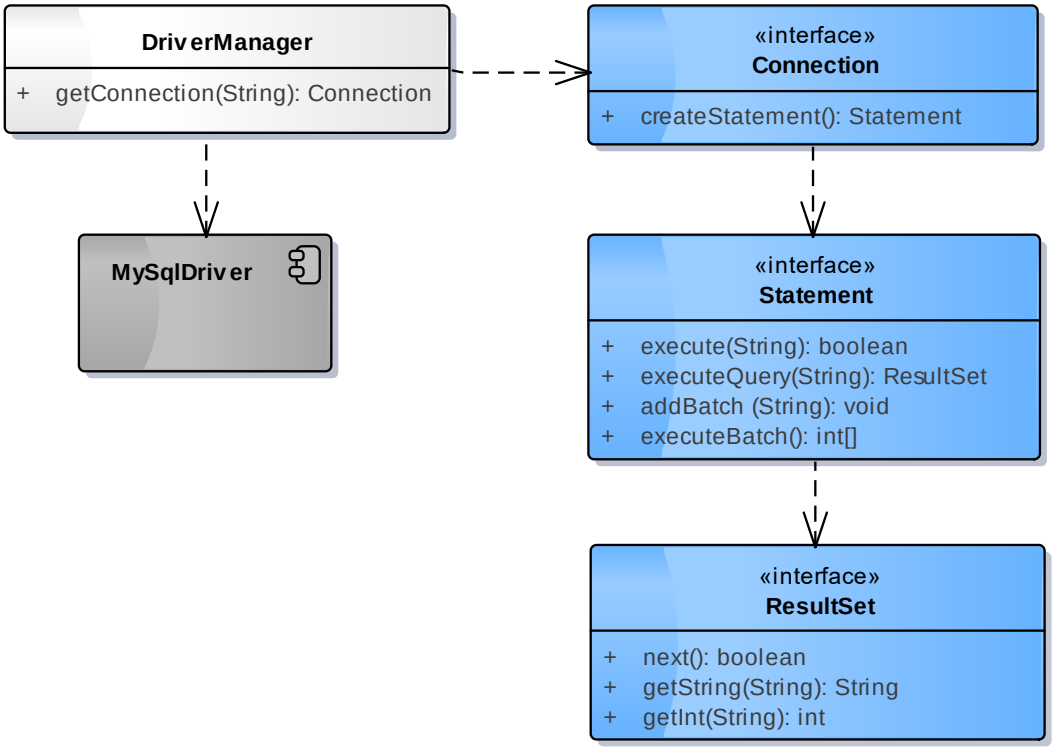
DateTimeFormatter

```
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



To execute SQL expressions using JDBC, first obtain a Connection object by passing the JDBC URL into the DriverManager's getConnection method. The syntax for URLs is jdbc:<subprotocol>:<subname>. Since the Connection interface extends Autocloseable, the "try-with-resources" construct can be used to automatically close the resource when the block exits.

Using the Statement object obtained from the connection, SQL commands to create and populate the Film table are executed as a batch. The array returned from executeBatch contains the number of rows affected by each command.

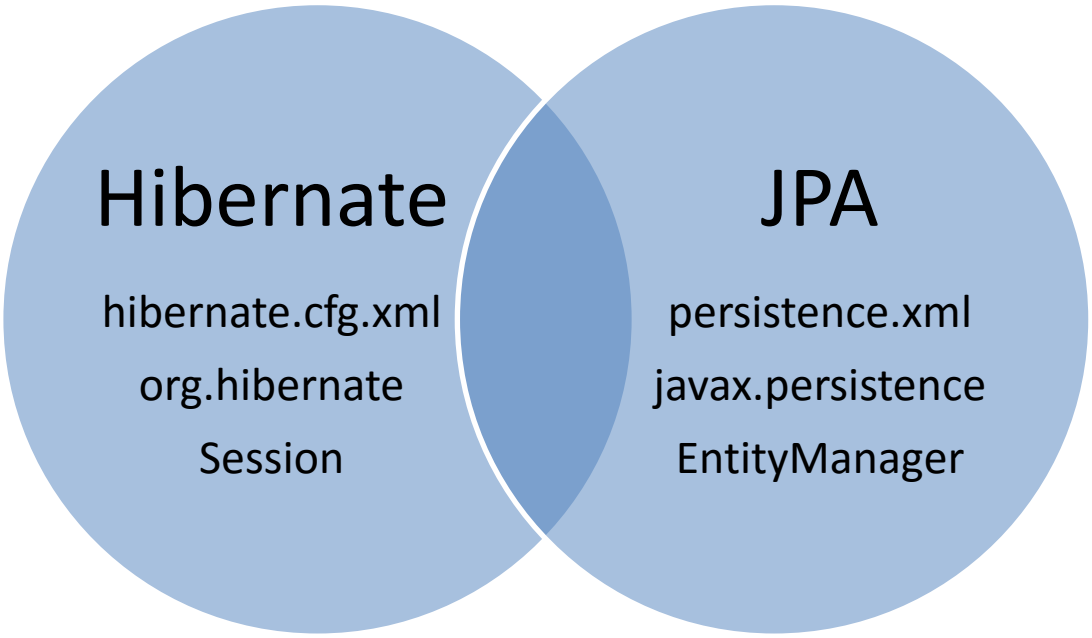
```
public class JpaFilmDAOIT {

	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 ...
	};

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

	@BeforeClass // runs once before any of the test methods
	public static void setup() {
		String url = "jdbc:mysql://localhost:3306/filmstore";
		try (Connection connection = DriverManager.getConnection(url,
                                                                 "root", "carpond")) {
			try (Statement statement = connection.createStatement()) {
				for (String command : commands) {
					statement.addBatch(command);
				}
				int[] updates = statement.executeBatch();
			}
		} catch (Exception ex) {
			throw new RuntimeException(ex);
		}
	}

	@Test
	public void selectByIdShouldReturnCorrectFilmFromStore() {
		// act
		Film film = dao.selectById(4L);
		// assert
		assertEquals("Pulp Fiction", film.getTitle());
	}
}
```

Without a database driver that's suitable for the subprotocol of the JDBC URL (mysql), the getConnection method will throw an SQLException. Add the following dependency to the build script.

```
dependencies {
    compile 'mysql:mysql-connector-java:5.1.34'
```

JPA

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

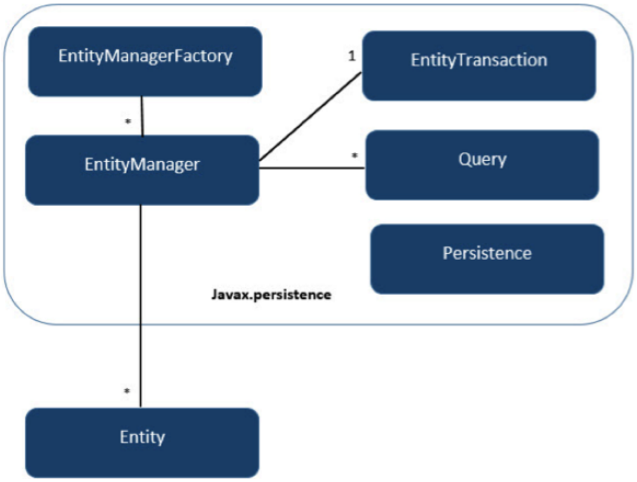


Object mapping

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:5.0.6.Final'
    compile 'org.hibernate:hibernate-java8:5.0.6.Final'
    compile 'javax.transaction:jta:1.1'
```

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

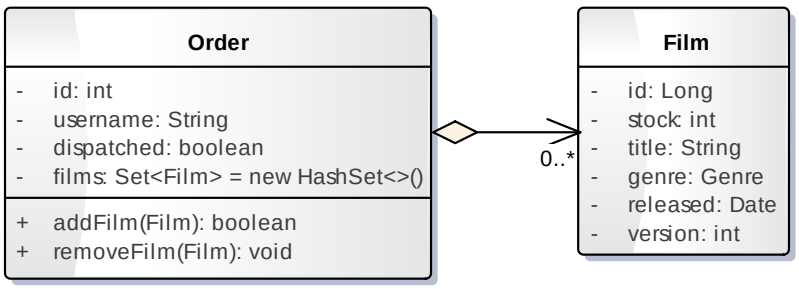
```
@Override
public Long insert(Film film) {
	Long id = 0L;
	EntityManager em = EntityManagerUtil.getEntityManager();
	em.getTransaction().begin();
	em.persist(film);// persists a transient instance
	id = film.getId();
	em.getTransaction().commit();// updates the database from the
	persistence context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

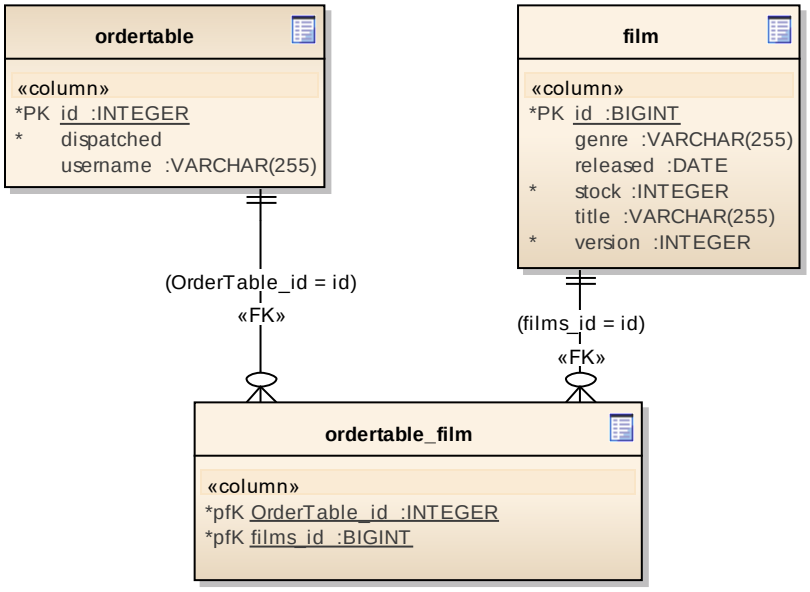
```java
/**
 * The delete method takes a persistent argument or a transient object with
 * an id matching an id in the database. The row in the database is deleted
 * when the transaction commits
 */
@Override
public boolean delete(Long filmId) {
       EntityManager em = EntityManagerUtil.getEntityManager();
       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();
       }
}
```

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.



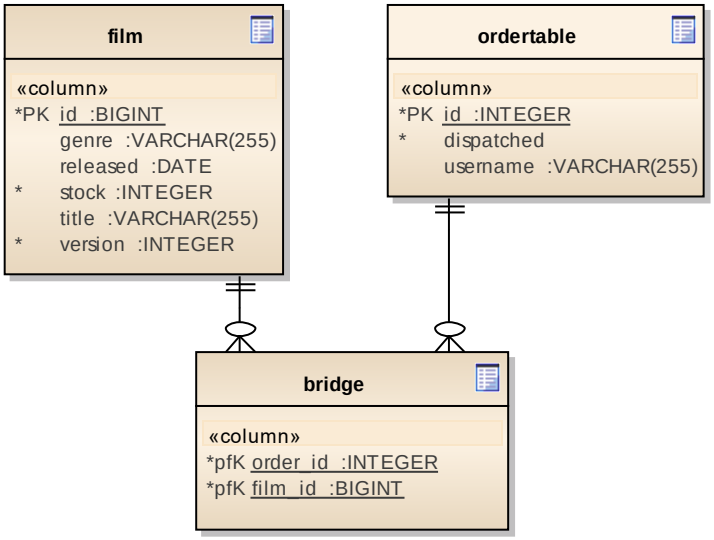

```
@Entity
@Table(name = "OrderTable")
public class Order {
	private boolean dispatched;

	@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:

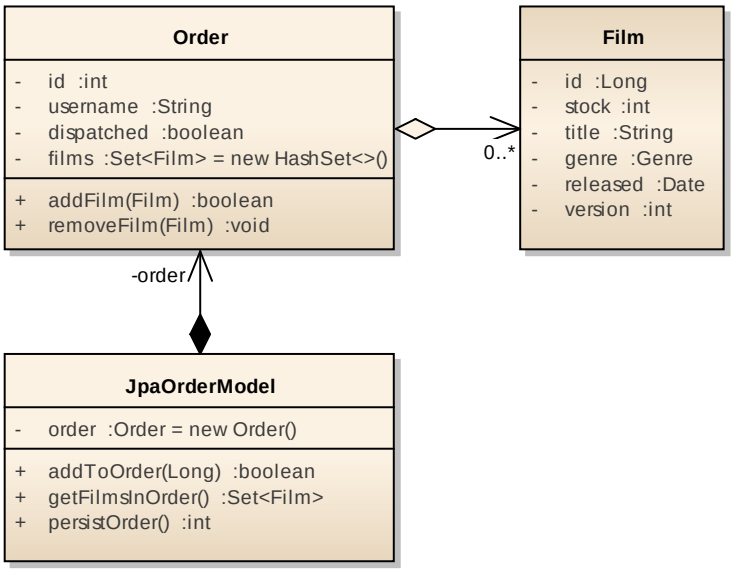
1. name is owningTable_associatedTable
2. joinColumns are the foreign key columns of the join table which reference the owning side of the association
3. inverseJoinColumns are the foreign key columns of the join table which reference the primary table of the entity that does not own the association

This would be written explicitly as follows:

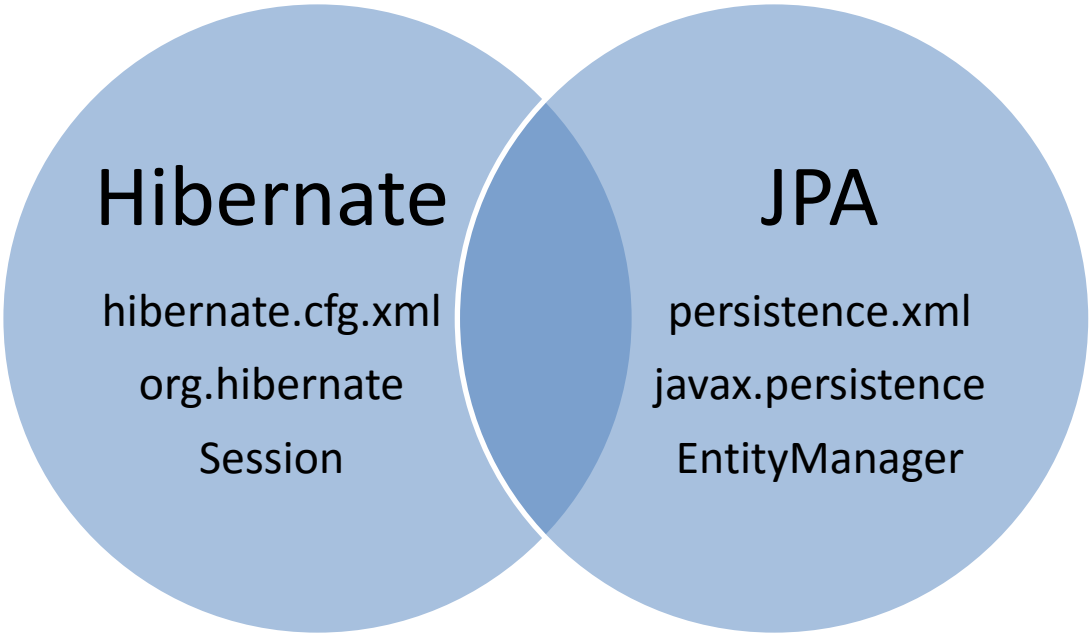
```
@JoinTable(name = "bridge",
           joinColumns = @JoinColumn(name = "order_id"),
           inverseJoinColumns = @JoinColumn(name = "film_id"))
@ManyToMany
private Set<Film> films = new HashSet<>();
```



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

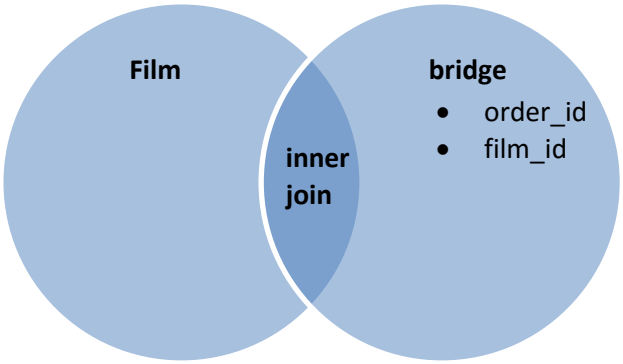
```
public class JpaOrderModelIT {
    // arrange
    private JpaOrderModel orderModel = new JpaOrderModel();

    @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.

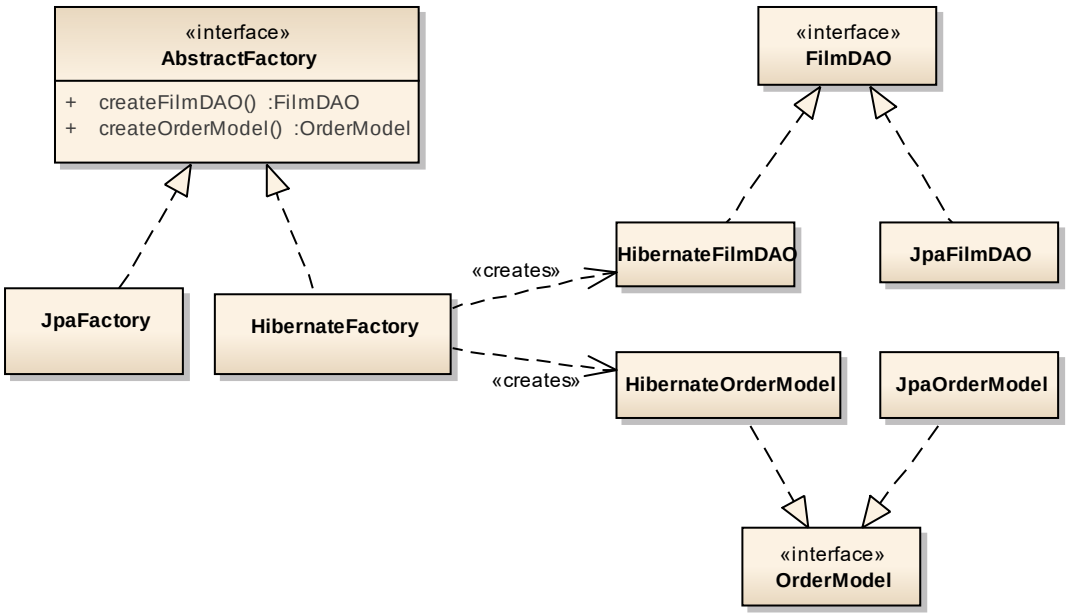
```
select title from Film
inner join bridge on Film.Id = bridge.film_id
where bridge.order_id = 1;
```

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

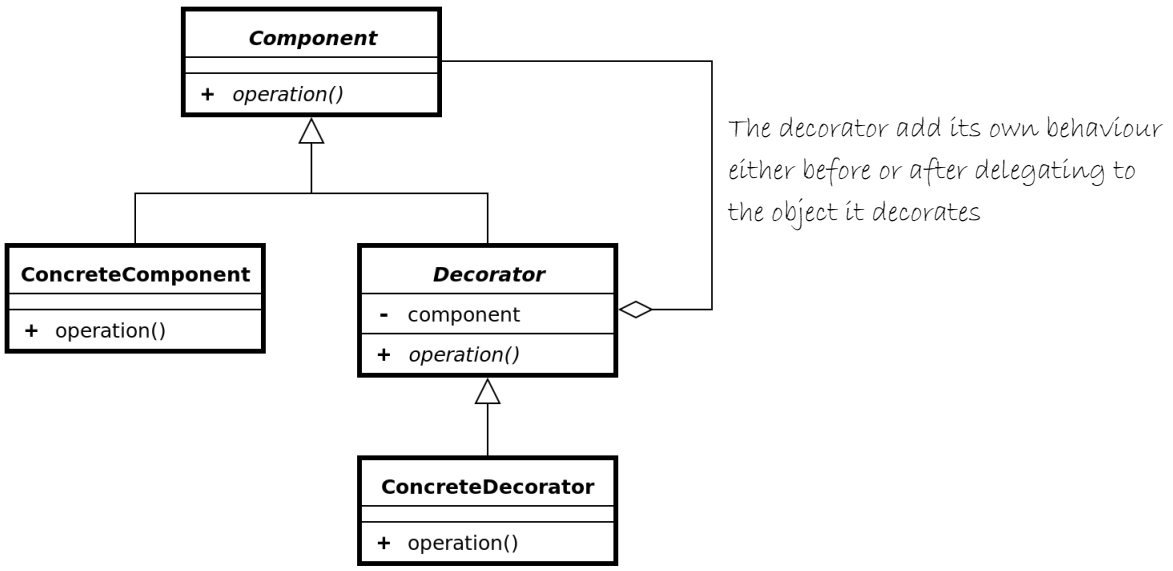
The abstract factory pattern provides an interface for creating families of related objects without specifying their concrete classes



Streams

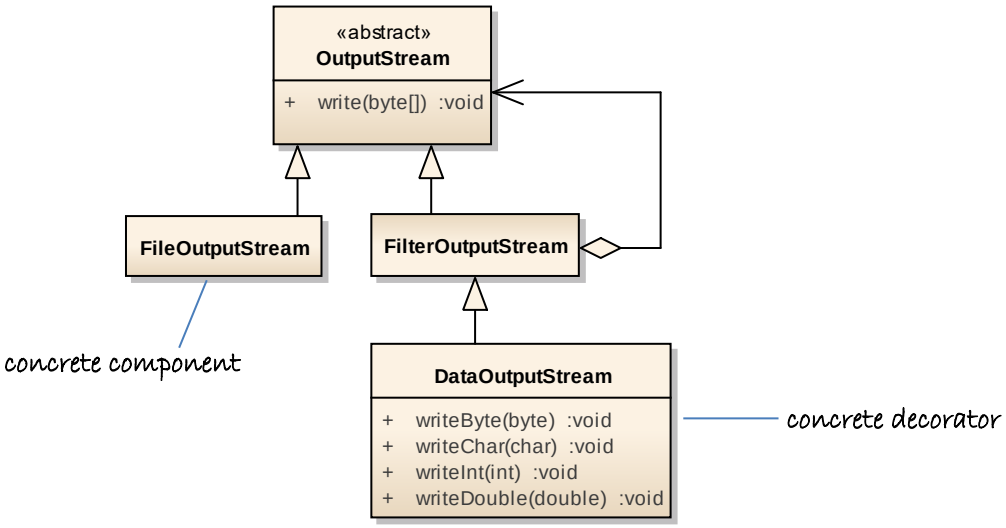
Decorator pattern

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**



The FileOutputStream can be decorated with a DataOutputStream

Writing primitives to a file

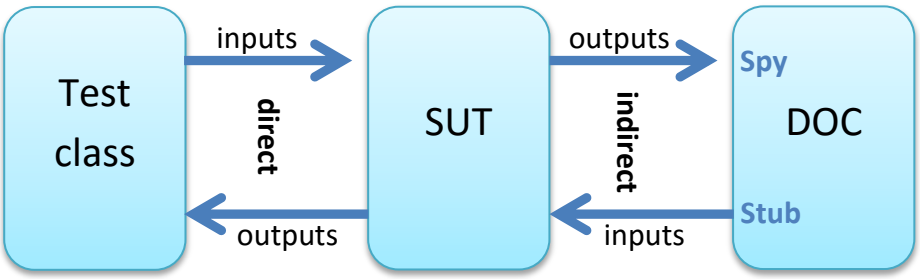
```
private static void writePrimitives() throws IOException {
    Path path = Paths.get("file.bin");
    try (DataOutputStream oos = new DataOutputStream(
                                Files.newOutputStream(path))) {
          oos.writeByte(127);       // 1 byte
          oos.writeChar('\u0061');  // 2 bytes
          oos.writeInt(2147483647); // 4 bytes
          oos.writeDouble(Double.MAX_VALUE); // 8 bytes
    }
}
```

Reading and writing text

The Files class contains static methods that operate on file and directories

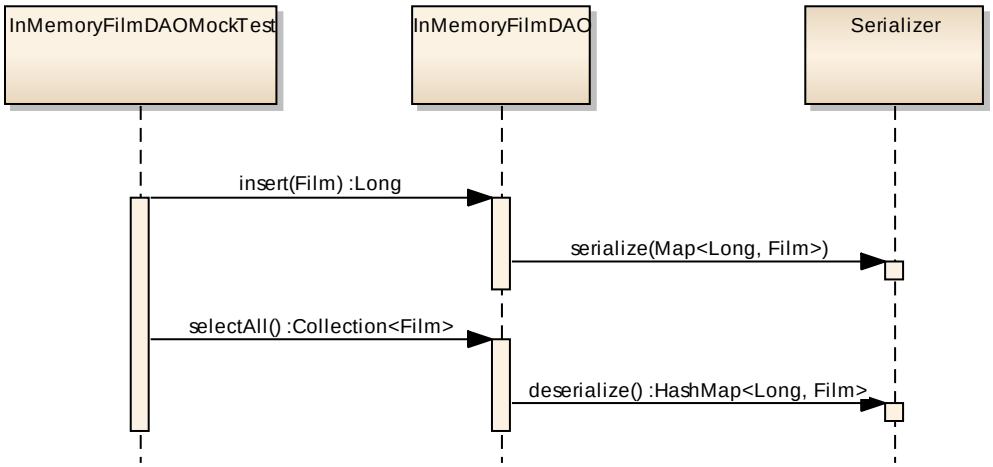
```
private static void writeText() throws IOException {
    Path path = Paths.get("file.txt");
    Set<String> zoneIds = ZoneId.getAvailableZoneIds();
    Files.write(path, zoneIds, StandardOpenOption.CREATE);
}

private static void readText() throws IOException {
    Path path = Paths.get("file.txt");
    List<String> lines = Files.readAllLines(path);
    lines.forEach(System.out::println);
}
```

Interactions testing

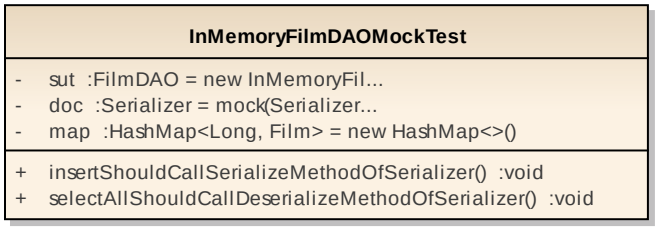


Unit tests apply to classes in isolation; the System Under Test (SUT) in the above diagram. They are intended to run fast and to pinpoint bugs with accuracy. **State testing** involves writing tests for direct inputs and outputs, while **interactions testing** verifies the way that the SUT interacts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

Test doubles look and behave like their release-intended counterparts, but are actually simplified versions. They're categorised into **spies**, which verify indirect outputs; **stubs**, which verify indirect inputs and **dummies**, which don't model interactions but can be passed from or into a method.



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

```
package session;

import static org.mockito.Mockito.*;
import java.util.*;
import org.junit.Test;

public class InMemoryFilmDAOMockTest {
   // arrange
   private Serializer doc = mock(Serializer.class);//test double
   private ConcurrentMap<Long, Film> map = new ConcurrentHashMap<>();
   private FilmDAO sut = new InMemoryFilmDAO(doc,map);

   //a spy is used to verify that the SUT calls the serialize method of the
   //collaborator (indirect output) when the insert method of the SUT is called
   @Test
   public void insertShouldCallSerializeMethodOfSerializer() {
       //arrange
       Film film = mock(Film.class); //dummy
       // act
       sut.insert(film);
       // assert
       verify(doc).serialize(map); //doc is a spy (verifies indirect outputs)
   }

   //a stub is used to verify that the deserialize method of the DOC returns a
   //Map (indirect input) when the selectAll method of the SUT is called
   @Test
   public void selectAllShouldCallDeserializeMethodOfSerializer() {
       //arrange (tell the DOC how it should act)
       when(doc.deserialize()).thenReturn(map);
       // act
       Collection<Film> films = sut.selectAll();
       // assert
       Map<Long, Film> map = verify(doc).deserialize();
                                   //doc is a stub (verifies indirect inputs)
   }
}
```

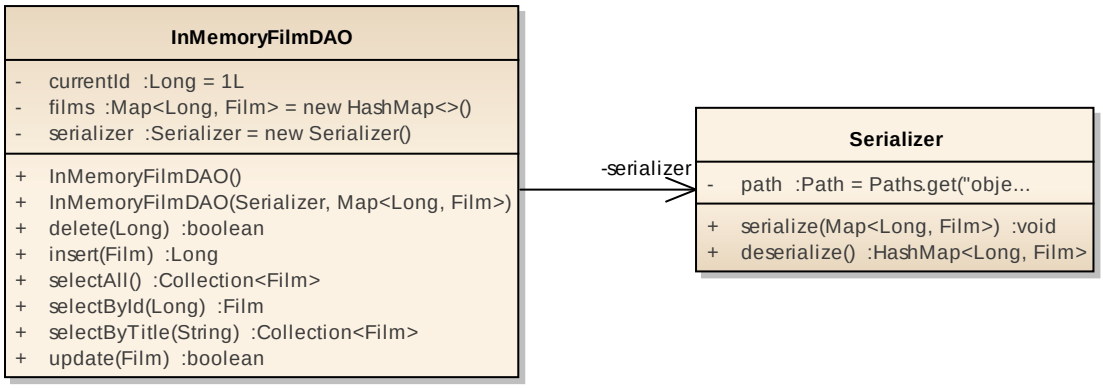
1. Add the Maven dependency for Mockito

```
<dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.9.5</version>
      <scope>test</scope>
</dependency>
```

2. Write the above test class and generate the Serializer class and methods in src/main/java folder
3. Run the InMemoryFilmDAOMockTest test
4. Complete the Serializer class
5. Run the InMemoryFilmDAOTest test

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 Serializer {

	private Path path = Paths.get("object.bin");

	public void serialize(ConcurrentMap<Long, Film> films) {
		try (ObjectOutputStream oos = new ObjectOutputStream(
				Files.newOutputStream(path))) {
			oos.writeObject(films);
		} catch (IOException e) {
			throw new FilmException(e.getMessage());
		}
	}

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

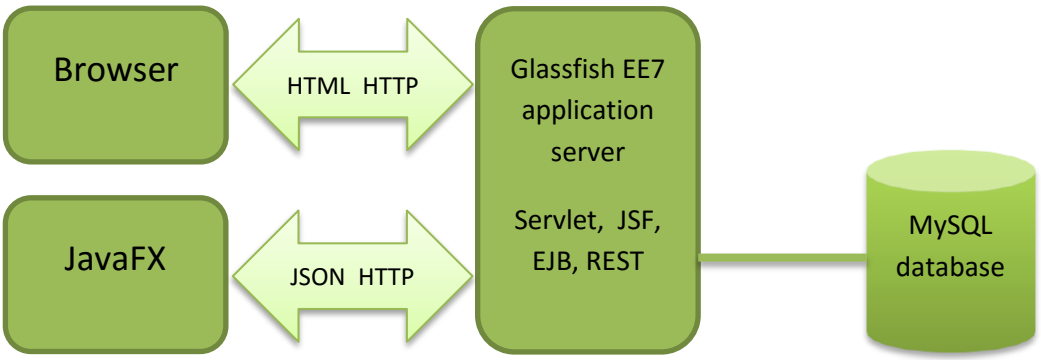
Serialized objects must implement the java.io.Serializable interface. To ensure matching versions of classes, including a serialVersionUID field in the serialized class is recommended. An InvalidClassException is thrown if the serial version of the class does not match that of the class descriptor read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

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

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. Hibernate 5 causes a NoSuchMethodError when deploying to glassfish. Fix this by replacing jboss-logging.jar with jboss-logging-3.3.0.Final.jar in \glassfish\modules

2. start server
   payara41\bin
   asadmin start-domain

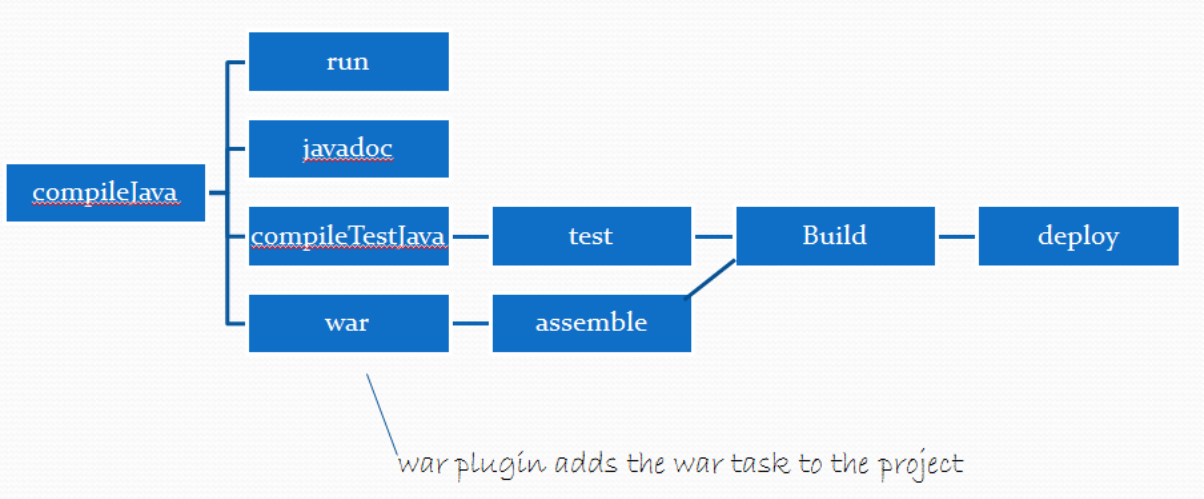
3. Logging
   olv.bat
   tail server.log at domains/domain1/logs/server.log

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

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

6. 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.

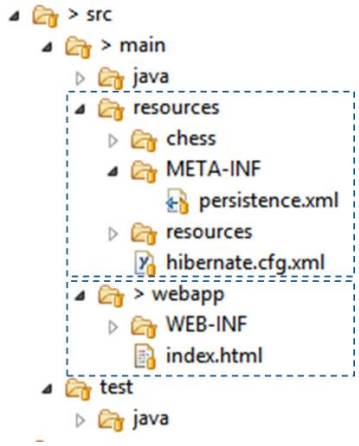
```
apply plugin:'application'
apply plugin: 'war'
```



7. 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'
```

8. 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>
```

9. Change hibernate hbm2ddl.auto property to validate, so tables aren't rebuilt
10. 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

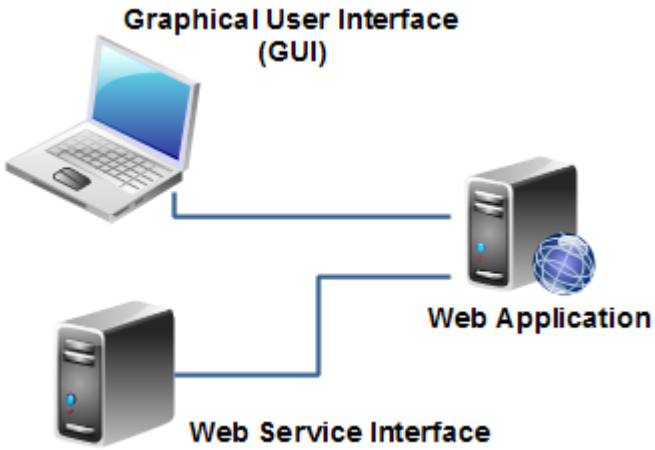
```
task deploy (type: Copy) {
    from 'build/libs'
    into '…domains/domain1/autodeploy'
    include '**/*.war'
}
deploy.dependsOn build
```

11. >gradle deploy generates the war file in the build/libs folder, asnd copies it to the glassfish autodeploy directory.
12. 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

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.

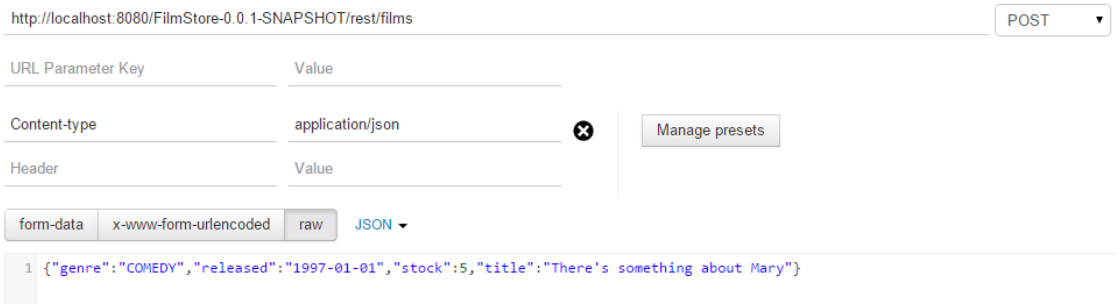
The Application class defines the components of a JAX-RS application and supplies additional meta-data. A JAX-RS application or implementation supplies a concrete subclass of this abstract class. The ApplicationPath annotation identifies the application path that serves as the base URI for all resource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

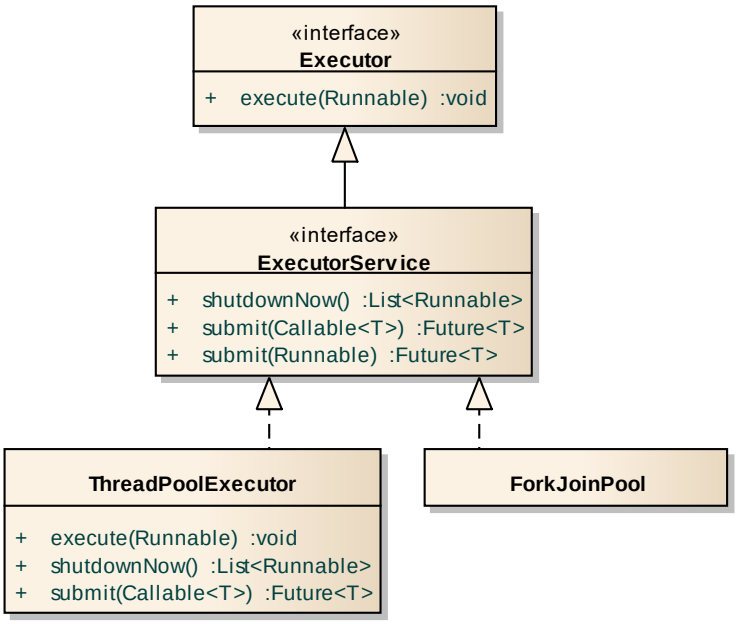
There are a number of libraries for encoding and decoding JSON. This is an example of using Google's Gson API. Currently the API can't deserialize dates, so set the Film class's LocalDate field as transient.

```
try {
	URL url = new URL(urlString);
	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();
	Film[] films = gson.fromJson(jsonString, Film[].class);
	return Arrays.asList(films);
} catch (Exception e) {
	throw new RuntimeException(e);
}
```

```
dependencies {
    compile 'com.google.code.gson:gson:2.3.1'
```

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

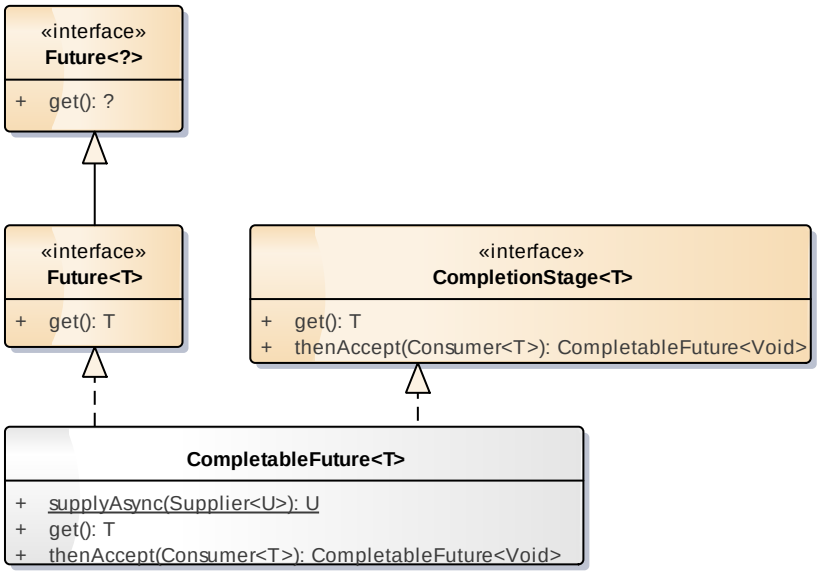
Thread pools consist of worker threads, which exist separately from the Callable tasks that are executed. Using worker threads minimizes the overhead due to thread creation.
A fixed thread pool always has a specified number of threads running; if a thread is terminated while it is still in use, it is 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 by 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().awaitQuiescence(5, TimeUnit.SECONDS);
```

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;
	}
}
```