

Java Programming Course for Featurespace

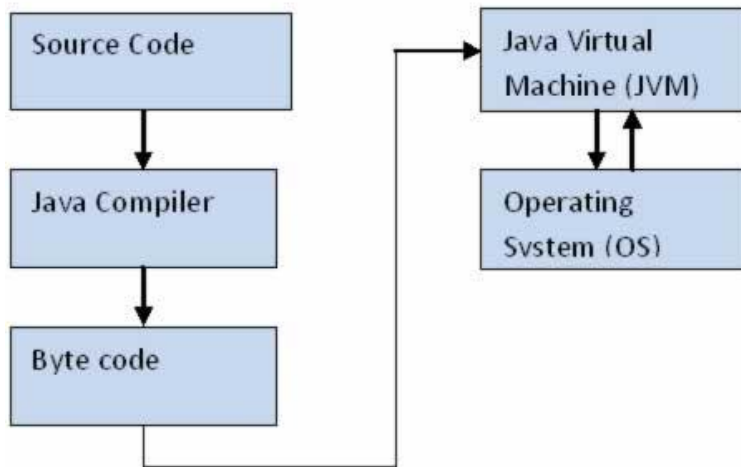


Contents

The Environment	2
Core Language Features.....	5
Classes and objects	9
Arrays and Strings.....	11
Static Methods	16
Inheritance	17
Exceptions	20
Interfaces.....	24
Collections	25
Maven.....	28
The Stream API	30
Test-driven development	33
Dates and Times	40
I/O	41
Thread safety.....	44
Concurrency	46
Featurespace topics	52
Databases	59
Design patterns.....	66
Projects.....	75
Sample exam questions	77

The Environment

Java Architecture



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

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

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

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

A simple command line application

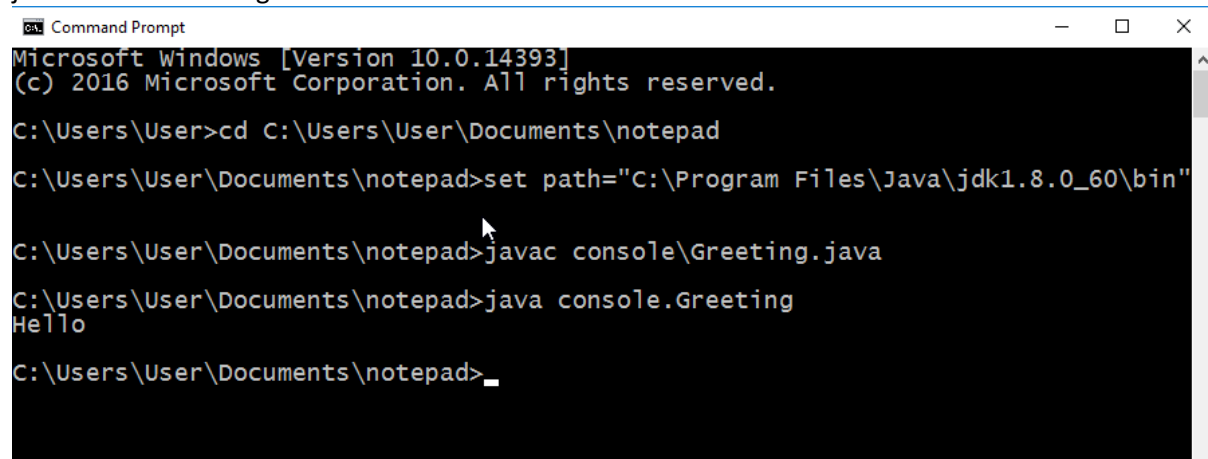
```
package console;
public class Greeting{
    public static void main(String[] args) {
        System.out.println("Hello");
    }
}
```

Compiling and running from the console

set path="C:\Program Files\Java\jdk1.8.0_60\bin"

javac console\Greeting.java

java console.Greeting

A screenshot of a Windows Command Prompt window. The title bar reads "Command Prompt". The window content shows the following text:
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\Users\User>cd C:\Users\User\Documents\notepad
C:\Users\User\Documents\notepad>set path="C:\Program Files\Java\jdk1.8.0_60\bin"
C:\Users\User\Documents\notepad>javac console\Greeting.java
C:\Users\User\Documents\notepad>java console.Greeting
Hello
C:\Users\User\Documents\notepad>_

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

Jar files

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

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

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

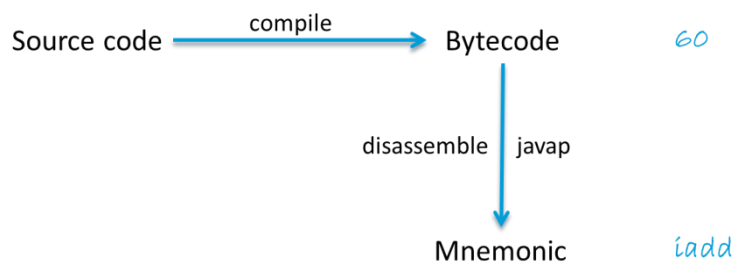
Extracting a jar file

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

Launching a jar file

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

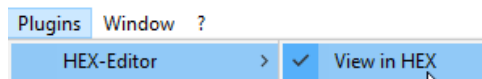
Bytecode



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

https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

Notepad++ has a Hex editor plugin



To view mnemonics in IntelliJ, install a plugin

File > Settings > Plugins > install ASM Bytecode Outline

Then right click a .class file and select Show Bytecode Outline

Software used on course

- JDK 8
 - Set JAVA_HOME in user environment variables
 - Path = %JAVA_HOME%\bin
- IntelliJ
- MySQL
 - user1, password
 - root, carpond
- MySQL workbench
- Postman HTTP debugger (Google chrome extension)
- Javadoc Documentation paths

File > Project Structure > Platform Settings > SDK
JavaSE: <http://docs.oracle.com/javase/8/docs/api>

File > Project Structure > Project Settings > Libraries
Java EE: <https://docs.oracle.com/javaee/7/api>
JUnit: <http://junit.sourceforge.net/javadoc/>

IntelliJ

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

New Project

File > New > Project > Java

File > Project structure > Project settings > Project language level

File > Project Structure > Platform Settings > SDK

<http://docs.oracle.com/javase/8/docs/api>

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

Short cuts

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

Basic Completion Ctrl+Space

Smart Completion Ctrl+Shift+Space.

Core Language Features

Primitive types

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

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

Conversion and casting

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

Operators

Increment operators

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

instanceof

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

Logical operators

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

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

Ternary operator

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

Operator precedence

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

Loops and Logic

Conditions

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

For loop

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

Break and continue

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

Labels and nested loops

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

Arithmetic series

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

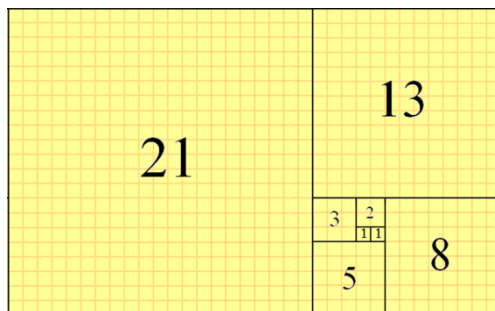
Geometric series

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

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


Fibonacci numbers

A tiling with squares whose side lengths are successive Fibonacci numbers



The sequence F_n of Fibonacci numbers is defined by

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

with seed values

$$F_1 = 1, F_2 = 1$$

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

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

Switch block

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

Classes and objects

Defining a class

```
package console;
public class Car {

    //state
    public int speed;
    private int gear;    Instance variables, accessible throughout the object

    //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;
    }
}
```

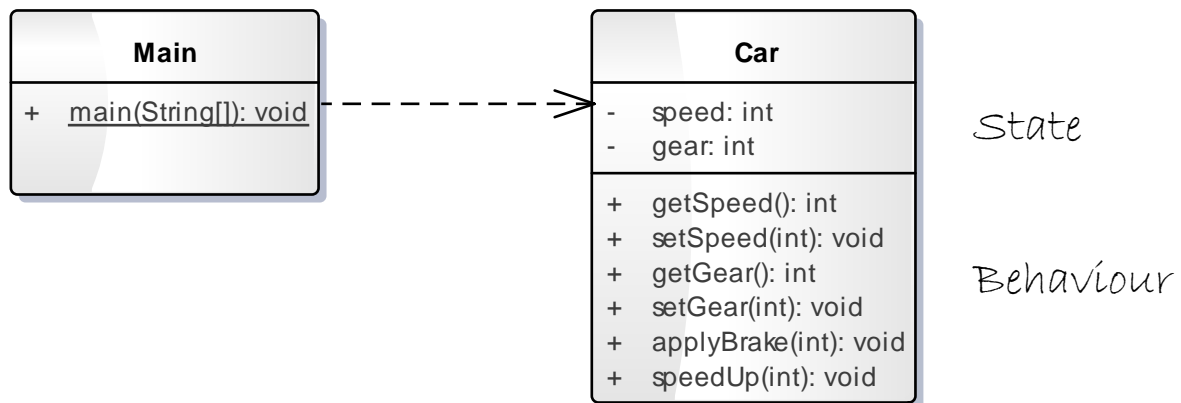
Instantiate a class

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

Access modifiers

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

UML Class Diagrams



Constructors

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

    public Car() {
    }

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

    // other methods
}
```

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

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

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

Arrays and Strings

Primitive arrays

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

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

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

Shortcut syntax

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

An array of objects

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

Strings

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

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

int index = quote.indexOf("unexamined"); //4
           quote.indexOf("Giraffe");    //-1

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

Mutable and immutable types

Strings are immutable

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

StringBuilder objects are mutable

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

Immutable objects have many advantages, including:

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

Immutable classes have

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

An immutable class

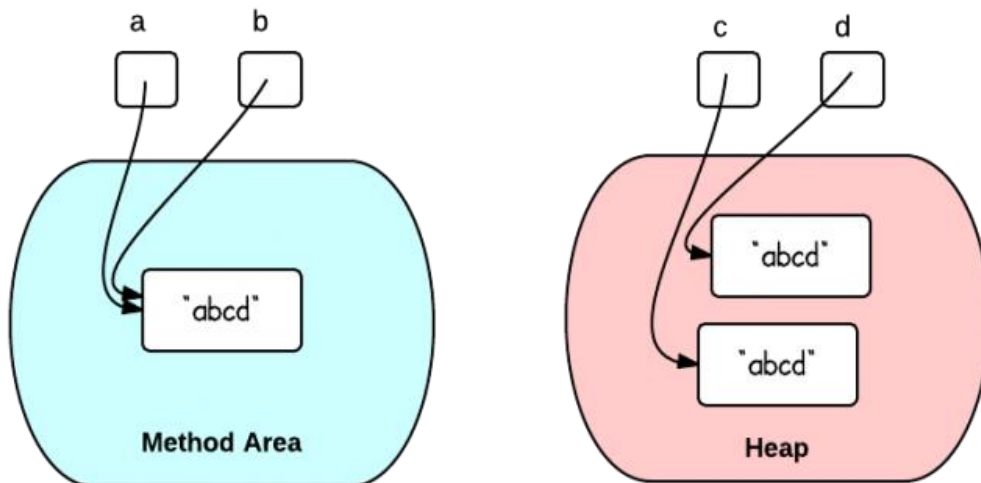
```
package java.time;  
public final class LocalDate {  
    private final int year;  
    private final short month;  
    private final short day;  
  
    //returns a copy of this LocalDate with the specified number of days added  
    public LocalDate plusDays(long daysToAdd) {  
        if (daysToAdd == 0) {  
            return this;  
        }  
        long mjDay = Math.addExact(toEpochDay(), daysToAdd);  
        return LocalDate.ofEpochDay(mjDay);  
    }  
}
```

A mutable class

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

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.

Keywords

abstract	continue	for	new	switch
assert	default	<i>goto</i> *	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
<i>const</i> *	float	native	super	while

Tweet

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

Enums

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

An enum is defined as follows

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

Using an enum in a class

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

Refactor > Encapsulate field

```
public class Class1 {  
    private DayOfWeek day;  
  
    public DayOfWeek getDay() {  
        return day;  
    }  
  
    public void setDay(DayOfWeek day) {  
        this.day = day;  
    }  
}
```

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

Using a switch block with an enum

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

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

Static Methods

Permutations & combinations

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

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

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

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

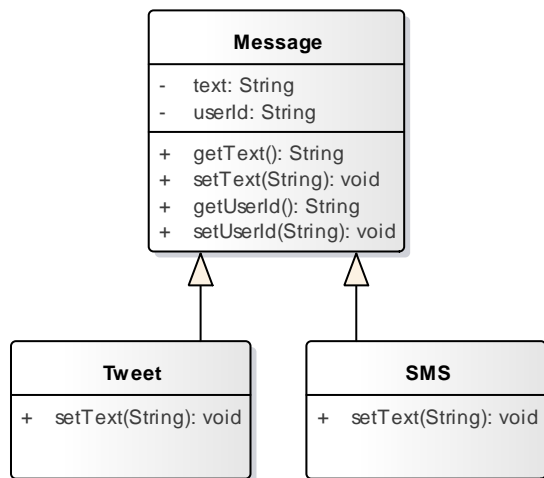
Factorial - The number of ways of arranging n distinct objects into an ordered sequence

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

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

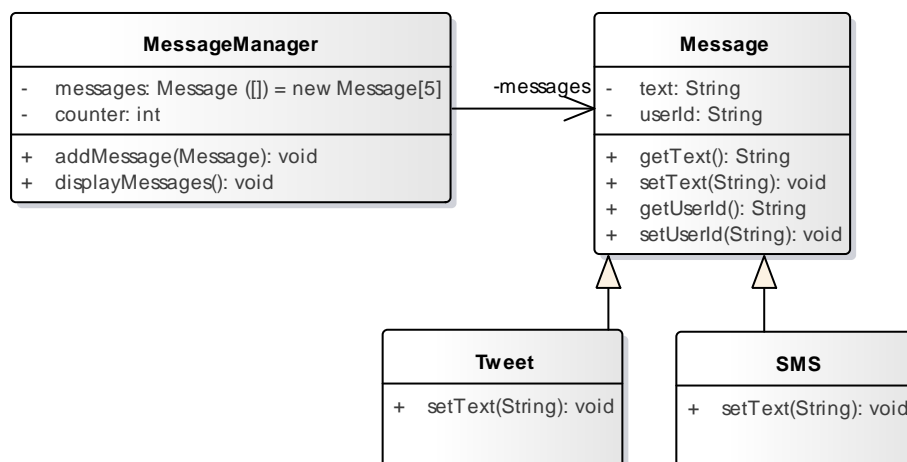
Inheritance

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

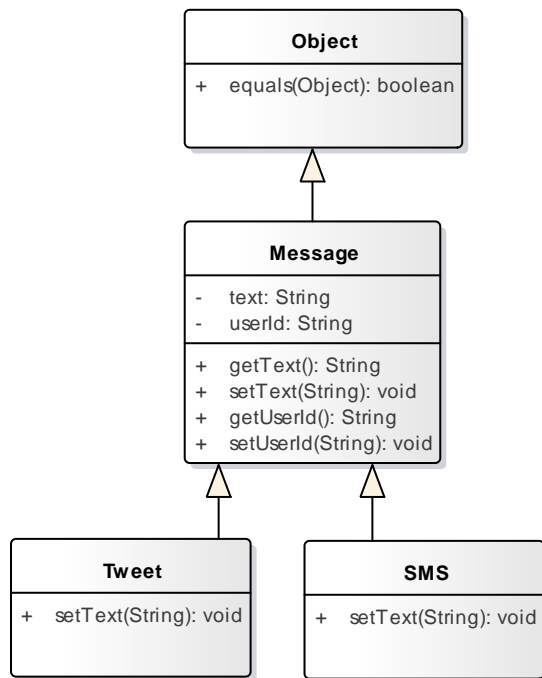


1. Tweet extends Message
2. Generate Message class
3. Refactor > Pull Members Up
4. Override `setText` method in Tweet class and move 140 character limit to this method
5. Call `super(userId, text)` from Tweet constructor
6. Generate the constructor in the base class
7. Build the SMS class with a 160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

Associations and generalisations



The Object class



Reference and Value Equality

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

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

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

Abstract classes

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
public abstract void setText(String text);
```

If a class includes abstract methods, then the class itself *must* be declared `abstract`, as in:

```
public abstract class Message {
```

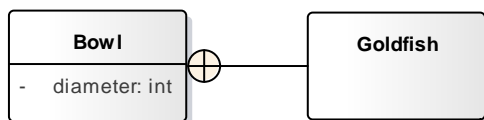
When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.

Consider using an abstract class if

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields

Inner classes

```
public class Bowl {  
    class Goldfish{  
  
    }  
}
```



A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class.

```
public static void main(String[] args) {  
    //An instance of InnerClass can exist only within an instance of outer class  
    Bowl bowl = new Bowl();  
    Bowl.Goldfish goldfish = bowl.new Goldfish();  
  
    //Static inner classes can exist independently of outer class  
    Customer.Address address = new Customer.Address();  
}
```

Benefits

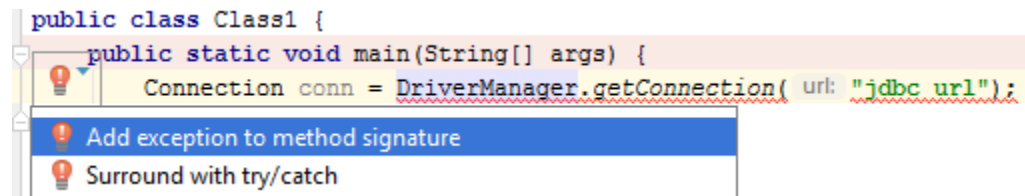
- If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.
- Increases encapsulation
- Can lead to more readable and maintainable code

Exceptions

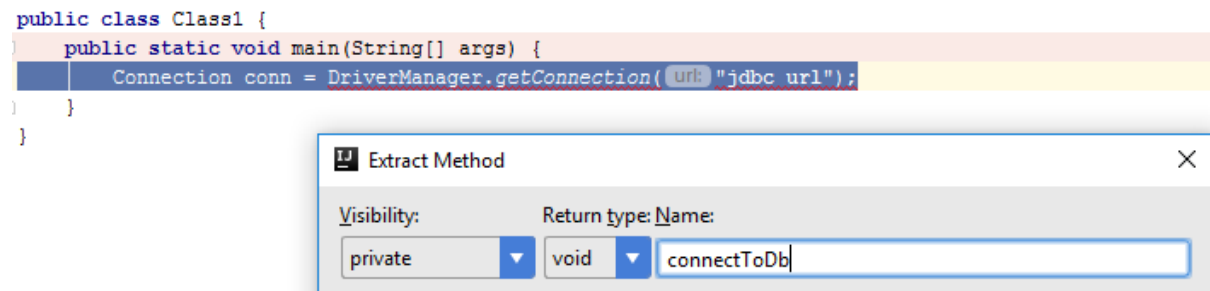
An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

A method can either handle an exception, or alternatively throw the exception object down the method call stack.

Extract method

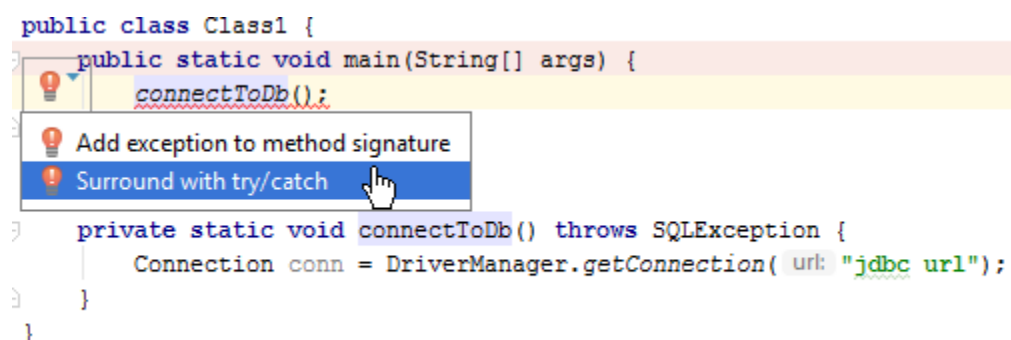


Select Refactor > Extract > Method



Handling exceptions

Surround the connectToDb method call with a try – catch block



Print the exception in the catch block

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

The method call stack

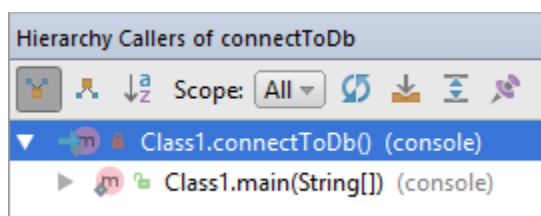
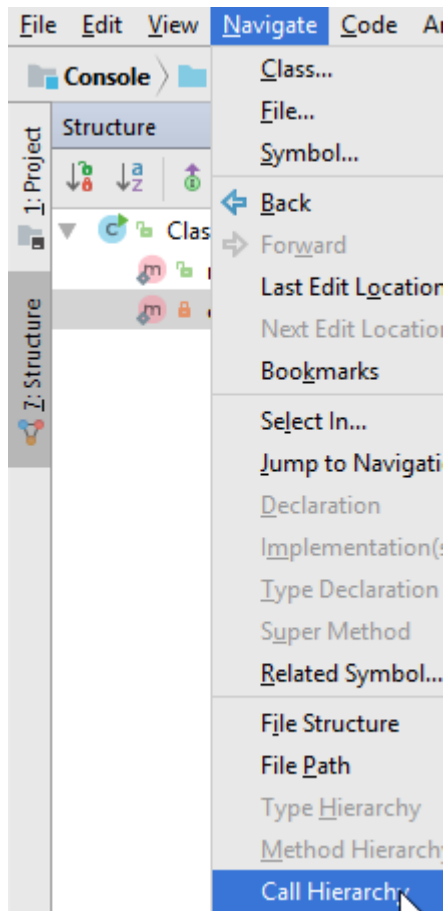
Display the structure View

View > Tool Windows > Structure

Display the Call hierarchy

Select the connectToDb method in the structure view

Navigate > Call Hierarchy



Multiple catch blocks

View the [JavaDoc for getConnection](#). Add a second catch block.

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

finally blocks

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

Autocloseable objects

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

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

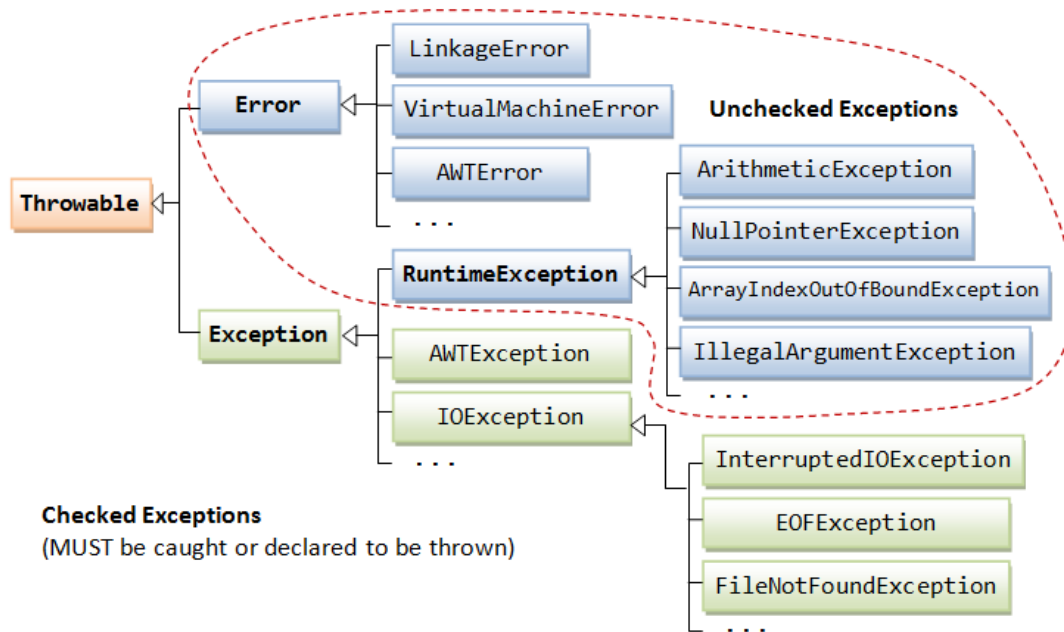
Unchecked exceptions

Dividing an int by zero causes an `ArithmeticException`

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

Handling `RuntimeExceptions` is optional

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



Exception Categories

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

Throwing an exception

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

```

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

```


User-defined Exceptions

```
public class MessageFormatException extends RuntimeException {  
    public MessageFormatException() {  
    }  
  
    public MessageFormatException(String message) {  
        super(message);  
    }  
}
```

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

Interfaces

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

Extract interface

Extract an interface from the MessageManager class and add an implementation

Extract Interface

Extract interface from:

console.MessageManager

☐ Extract interface

☒ Rename original class and use interface where possible



Rename implementation class to:

ArrayMessageManager

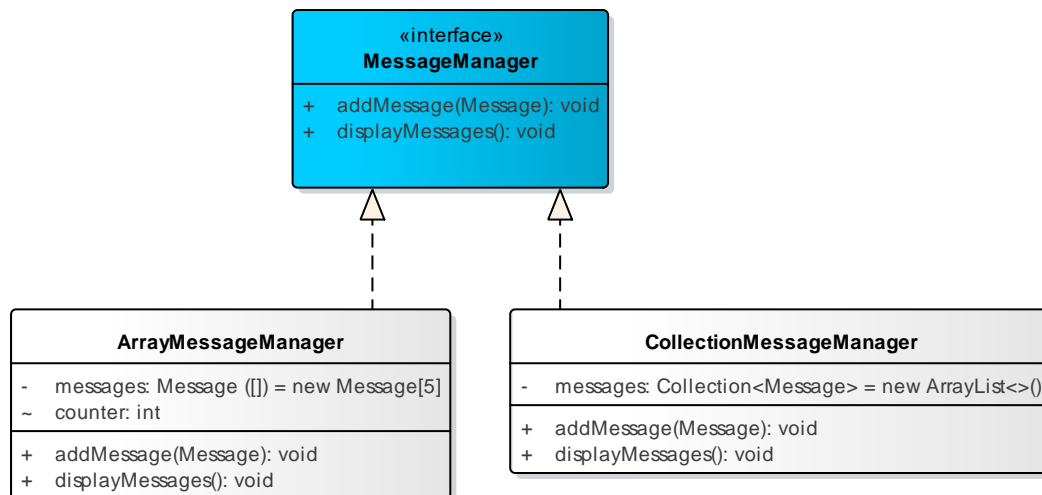
Package for original class:

console

Members to form interface

	Member	Make abstract
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>  addMessage(message:Message):void	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>  displayMessages():void	<input checked="" type="checkbox"/>

Java



```

public interface MessageManager {
    void addMessage(Message message);

    void displayMessages();
}

```

Collections

Generic class

A generic class is parameterized over types

```

package example;

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

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

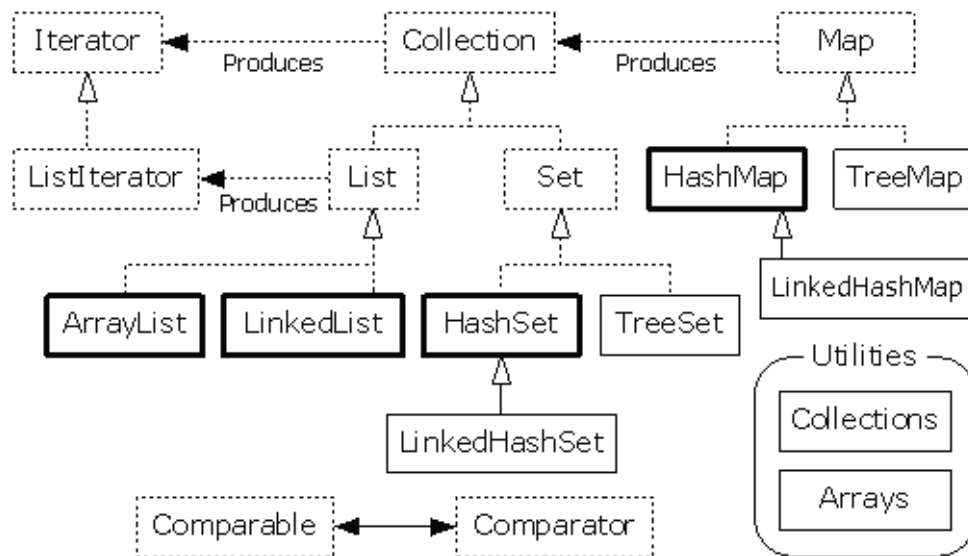
MyCollection<Message> myCollection = new MyCollection<>();
myCollection.add(t1);
myCollection.add(t2);

```

Type Parameter Naming Conventions

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

Collections framework



Interfaces in the collection hierarchy

The [java.util](#) package contains the collections framework

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

ArrayList

```
public class CollectionMessageManager implements MessageManager {  
  
    private Collection<Message> messages = new ArrayList<>();  
    @Override  
    public void addMessage(Message message) {  
        messages.add(message);  
    }  
  
    @Override  
    public void displayMessages() {  
        for (Message message: messages) {  
            System.out.println(message.getText());  
        }  
    }  
}
```

Implementing Iterable

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

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

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

    }

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

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

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

Maven

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

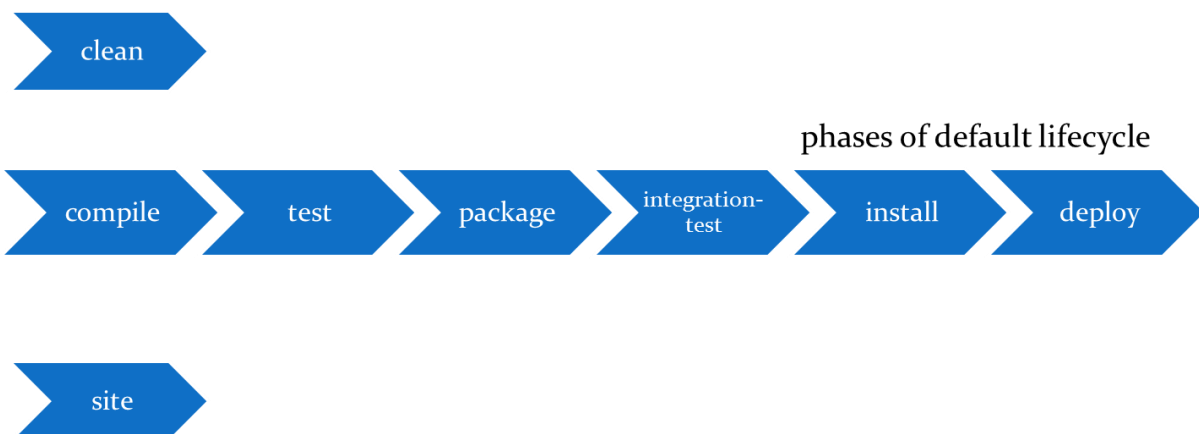
For the majority of users, Maven is a build tool that compiles, tests and packages deployable artifacts from source code.

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

The Build Lifecycle

A lifecycle is an organized sequence of phases that exist to give order to a set of goals. There are three standard lifecycles in Maven: clean, default and site.

<http://books.sonatype.com/mvnref-book/reference/lifecycle.html>



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

Default Goals for JAR Packaging

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

POM

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

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

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

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

The Stream API

Lambda Expressions

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

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

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

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

        List<Message> filteredMessages = //

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

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

Functional interfaces

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

Right click the lambda expression > Refactor > Extract > Variable

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

Anonymous inner classes

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

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

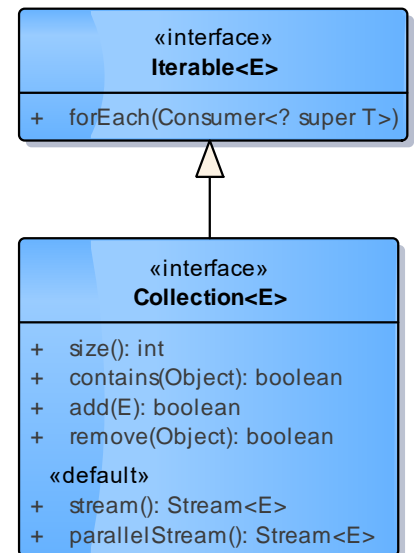
Default methods

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

```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action) {
        //implementation
    }
}

public interface Collection<E> extends Iterable<E> {
    int size();
    boolean contains(Object o);
    boolean add(E e);
    boolean remove(Object o);

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



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

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

Collecting results

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

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

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

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

Chaining methods

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


Method references

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

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

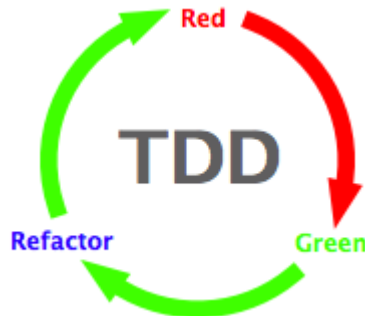
There are four kinds of method references:

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

```
List<Message> filteredMessages = messages.  
    stream().  
    filter(messagePredicate).  
    collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

Test-driven development

Overview



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

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

Benefits

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

Unit tests.

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

Integration tests.

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

Acceptance tests.

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

Phases when writing a test

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

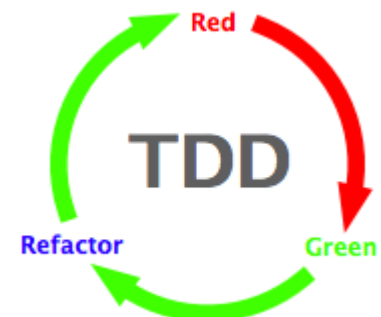
```
package entity;
import static org.junit.Assert.*;
import java.time.LocalDate;
import org.junit.Test;

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

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

TDD Rhythm

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



Apply this to the above example

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

Unit tests and expected exceptions

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

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

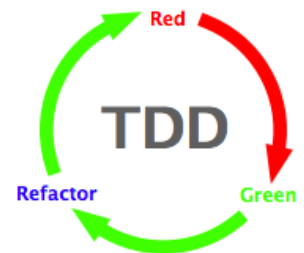
Following the TDD rhythm

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

Override equals

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

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



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

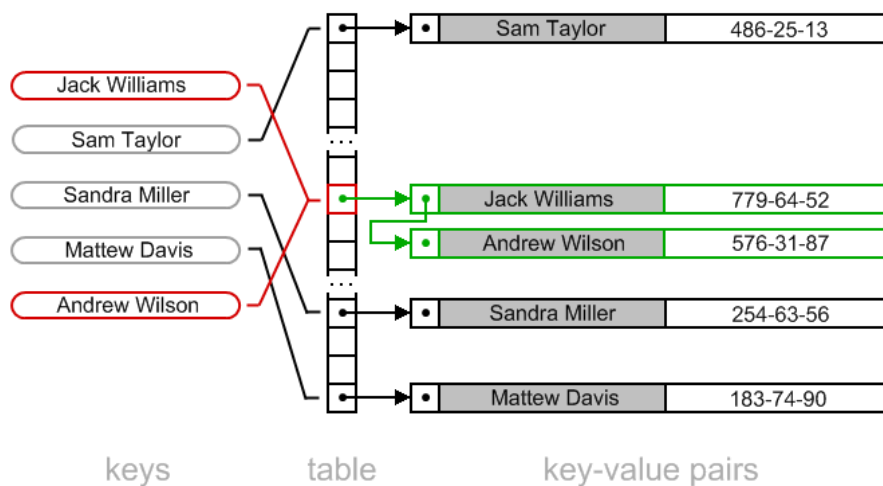
```
@Test
public void filmsWithSameTitleShouldBeEqual() {
    // arrange
    Film film1 = new Film();
    film1.setTitle("The Godfather");
    Film film2 = new Film();
    film2.setTitle("The Godfather");

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

Override hashCode

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

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



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

    Set<Film> films = new HashSet<>();
    //act
    films.add(film1);
    films.remove(film2);
    //assert
    assertEquals(0, films.size());
}
```

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

String formatting

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

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

Collections classes

```
public class CollectionTest {

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

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

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

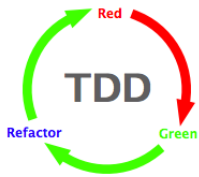
    @Test
    public void mapContainsUniqueKeys() {
        //arrange
        Map<Long, Film> map = new HashMap<>();
        //act
        Film previousValue1 = map.put(1L, film1);
        Film previousValue2 = map.put(1L, film2);
        //assert
        assertNull(previousValue1);
        assertEquals(film1,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

```

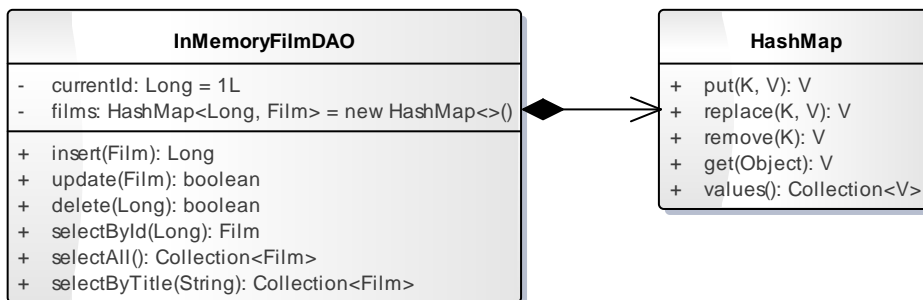
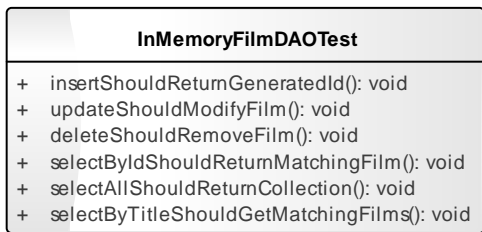
@Test
    public void addUpdateAndRemoveFromMap() {
        //arrange
        Map<Long, Film> map = new HashMap<>();
        //act
        Film previousValue1 = map.put(1L, film1); //add key and value to a map
        Film previousValue2 = map.replace(1L, film2); //null if key isn't in map
        Film removedFilm = map.remove(1L); //remove value with specified key
        //assert
        assertNull(previousValue1);
        assertEquals(film1, previousValue2);
        assertEquals(film2, removedFilm);
        assertTrue(map.isEmpty());
    }
}

```

Data Access Object



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



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

InMemoryFilmDAO

Complete the `selectByTitle` method, using a lambda expression.

```
public class InMemoryFilmDAO {  
    private Long currentId = 1L;  
    private HashMap<Long, Film> films = new HashMap<>();  
    public Collection<Film> selectByTitle(String search) {  
        Collection<Film> filmCollection = films.values();  
    }  
}
```


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 zonedDateTime1 = ZonedDateTime.of(localDateTime1,
                                                  ZoneId.of("Europe/Berlin"));
System.out.printf("ZonedDateTime %s\n", zonedDateTime1);
ZonedDateTime zonedDateTime2 = zonedDateTime1.plusHours(24);
System.out.printf("Summer time %s\n", zonedDateTime2);
```

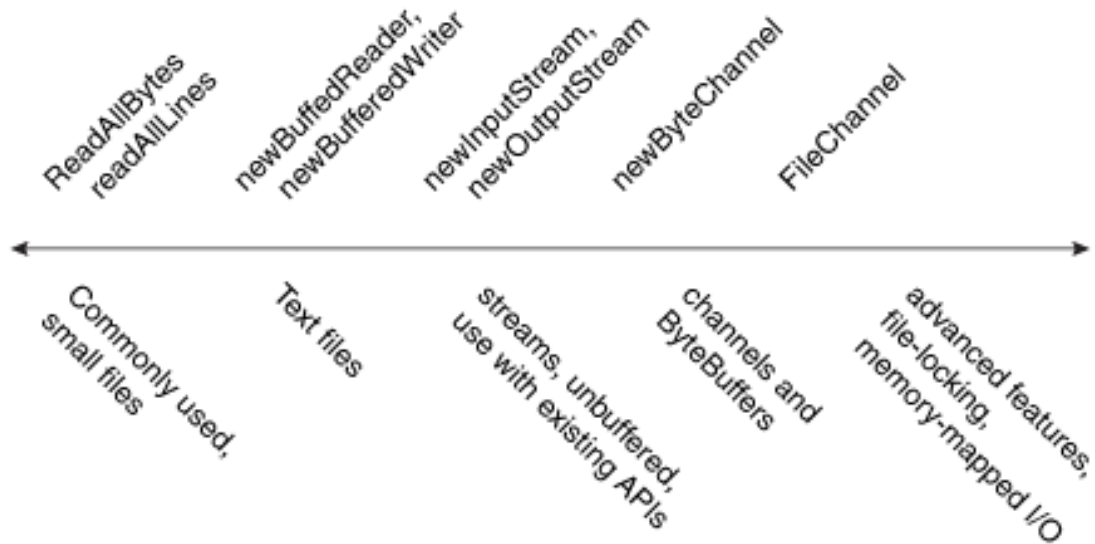
DateTimeFormatter

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL);
System.out.printf("Formatted: %s\n", formatter.format(zonedDateTime2));
```

I/O

Reading, Writing, and Creating Files

This diagram arranges the file I/O methods by complexity



Small Files

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

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

Text Files

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

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

Unbuffered Streams

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

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

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

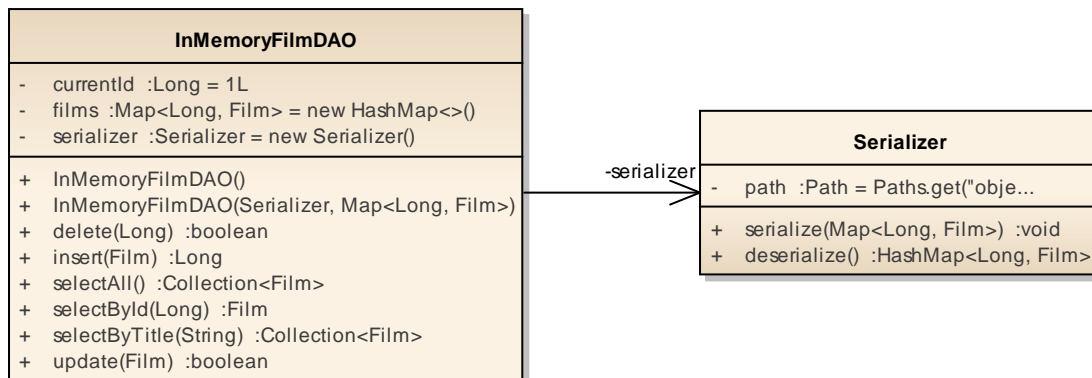
Serializable interface

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

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

Serialization

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



```

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

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

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

Thread safety

```
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;
    }
}
```

Synchronized methods

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

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

Synchronized block

```
public class InMemoryFilmDAO implements FilmDAO {

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

Generating a sequential id

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

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

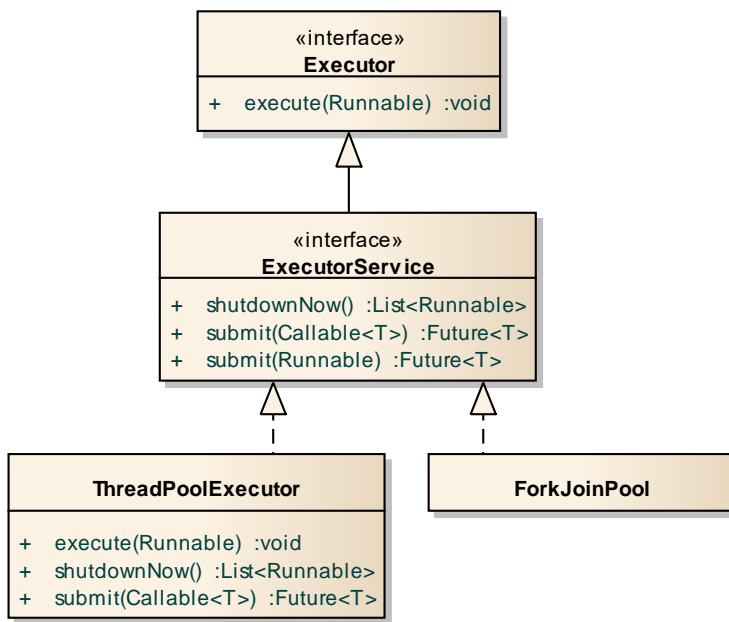
Concurrency

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

Executors



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

Using an Executor

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

Thread Pools

Thread pools consist of worker threads, which exist separately from the `Callable` tasks that are executed. Using worker threads minimizes the overhead due to thread creation.

A fixed thread pool always has a specified number of threads running; if a thread is terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

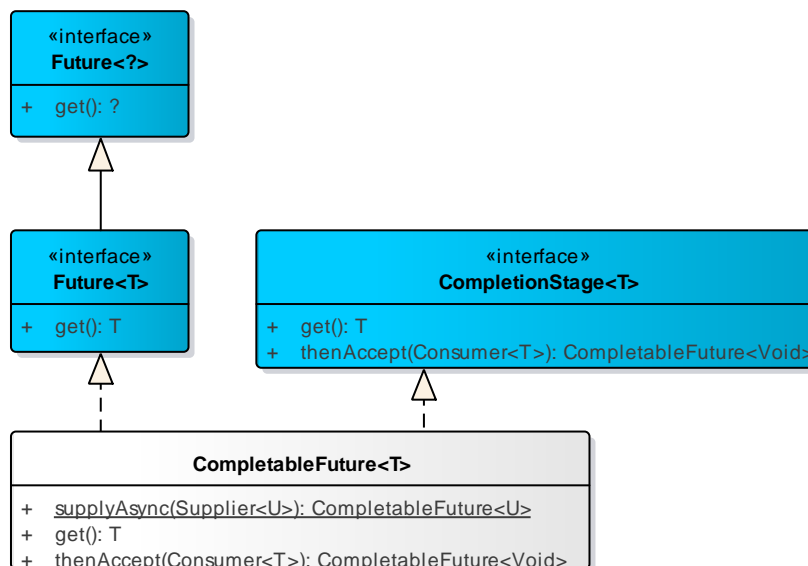
An expandable thread pool is suitable for applications that launch many short-lived tasks.

The Future interface

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

Asynchronous methods

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



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

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

Parallel tests

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

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

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

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

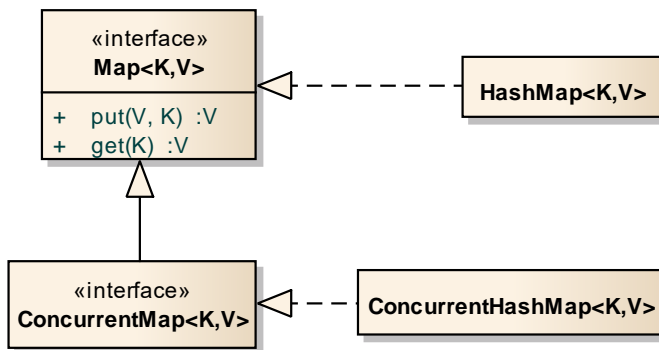
    @Test
    @Repeating(repetition = 1000) //execute method 1000 times
    @Concurrent(count = 5) //in 5 threads
    public void insertFilmsConcurrently() {
        sut.insert(new Film());
    }

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

This requires the following dependency

```
<dependency>
  <groupId>com.google.code.tempus-fugit</groupId>
  <artifactId>tempus-fugit</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
```

java.util.concurrent



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

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

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

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

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

Thread interference

Source code

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

Corresponding mnemonic, bytecode and description

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

Viewing Account.class in a binary editor

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

java.util.concurrent.atomic

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

Featurespace topics

Style guide

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

Jackson

<https://github.com/FasterXML/jackson>

<http://fasterxml.github.io/jackson-databind/javadoc/2.2.0/>

<http://www.baeldung.com/jackson-object-mapper-tutorial>

```
public class Serialize {
    public static void main(String args[]) {
        try {
            //Serialize to JSON
            Date date = new GregorianCalendar(1997, 4, 27).getTime();
            Film film = new Film("There's something about Mary", 2, date,
                                Genre.COMEDY);

            ObjectMapper mapper1 = new ObjectMapper();
            String jsonString = mapper1.writeValueAsString(film);
            System.out.printf("Serialized object: %s\n", jsonString);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public class Deserialize {
    public static void main(String args[]) {
        try {
            URL url = new URL("http://sdineen.uk/api/films/");
            ObjectMapper mapper = new ObjectMapper();
            mapper.configure(MapperFeature.ACCEPT_CASE_INSENSITIVE_PROPERTIES,
                             true);

            //Film [] films = mapper.readValue(jsonString, Film[].class);
            Film [] films = mapper.readValue(url, Film[].class);
            for (Film film:films) {
                System.out.println(film.getTitle());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

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

Hamcrest

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

```
public class FilmTest {

    @Test
    public void constructorShouldInitialiseFields() {
        // arrange and act
        Film film = new Film("The Pink Panther", 5, LocalDate.of(1964, 1, 20),
                                Genre.COMEDY);

        Long id = film.getId();
        String title = film.getTitle();
        int stock = film.getStock();
        LocalDate released = film.getReleased();
        Genre genre = film.getGenre();

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

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

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

Add the following Javadoc link to the Hamcrest-core dependency in File > Project Structure > Libraries

<http://hamcrest.org/JavaHamcrest/javadoc/1.3/>

`nullValue()` is a method of `Matchers`; see

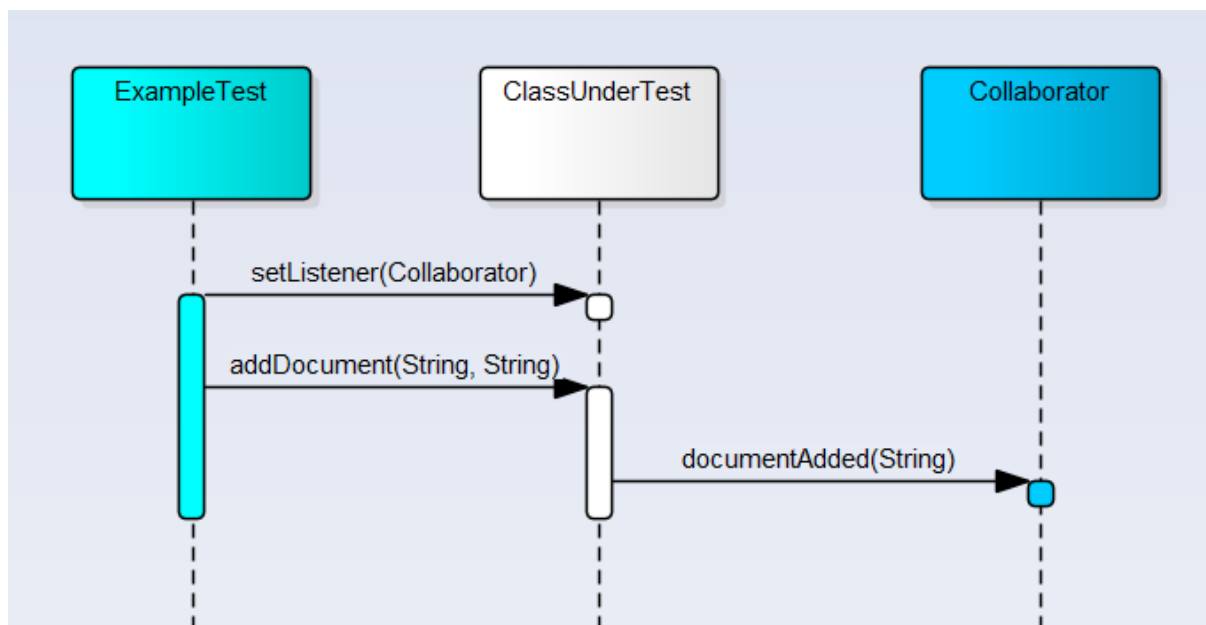
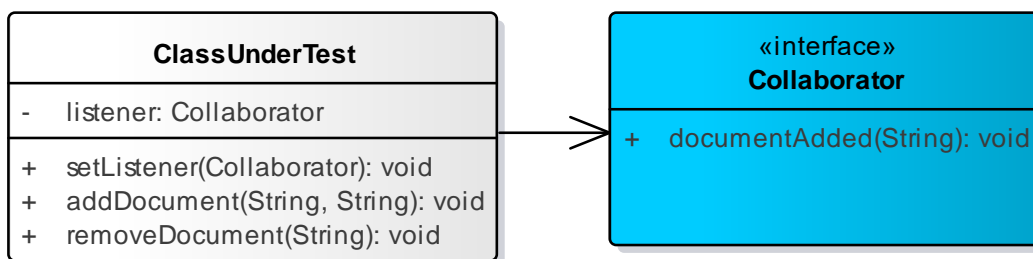
<http://hamcrest.org/JavaHamcrest/javadoc/1.3/index.html?org/hamcrest/Matcher.html>

EasyMock

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

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

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



ExampleTest

```
package featurespace.easymock;

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

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

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

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

ExampleTest with annotations

```
public class ExampleTestWithAnnotations extends EasyMockSupport {

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

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

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

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


InMemoryFilmDAOTest

```
package featurespace.easymock;

public class InMemoryFilmDAOTest extends EasyMockSupport {

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

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

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

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

InMemoryFilmDAOTest with Annotations

```
public class InMemoryFilmDAOTestWithAnnotations extends EasyMockSupport {

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

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

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

    //Test methods as above
}
```

Javadoc

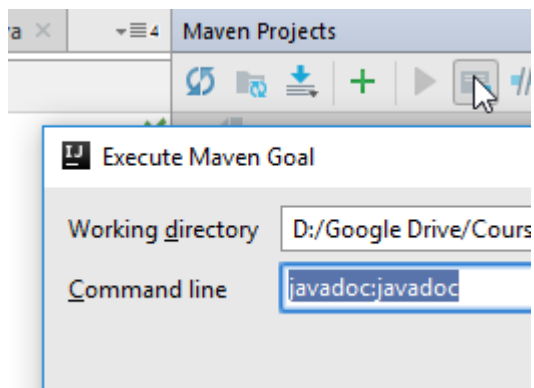
<https://maven.apache.org/plugins/maven-javadoc-plugin/>

Add the plugin to the reporting section of the POM

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

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

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



Style guide <https://google.github.io/styleguide/javaguide.html#s7-javadoc>

Guava

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

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

Immutable collections

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

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

Apache Commons

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

Math

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

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

CSV

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

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

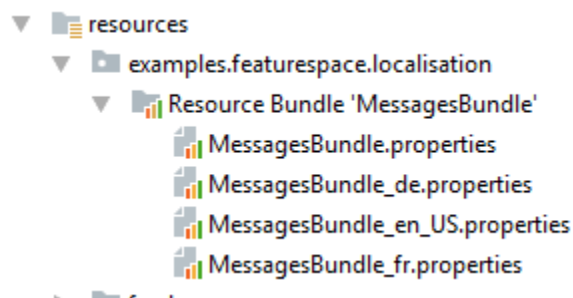
Localisation

```
package examples.featurespace.localisation;

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

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

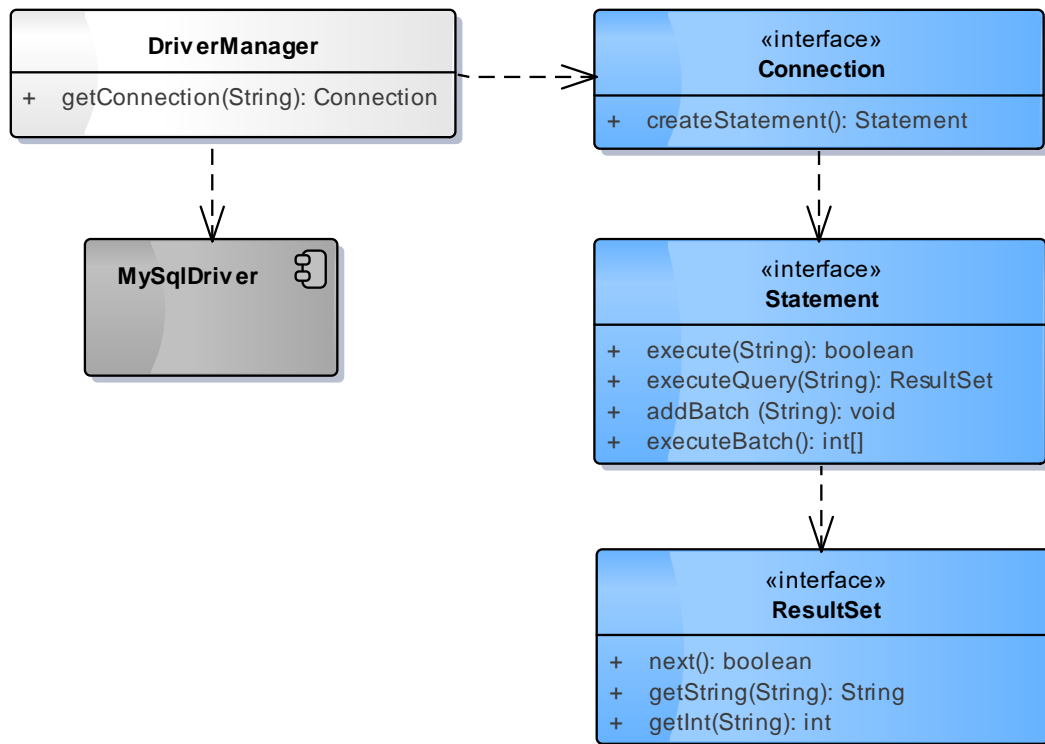
Add properties files to the resources folder



Databases

JDBC

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



SQLite

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

```

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

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

```
public class JdbcFilmDAOIT {

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

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

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

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

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

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

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

Pessimistic concurrency

Transaction Isolation Levels

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

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

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

Optimistic concurrency

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

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

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

Add a version field of type int to the Film class

Partial implementation

```
public class JdbcFilmDAO implements FilmDAO {

    private String url = "jdbc:mysql://localhost:3306/filmstore";
    private String username = "root";
    private String password = "carpond";

    @Override
    public Long insert(Film film) {
        String sql = "insert into Film (title, stock, released, genre, version)
                      values (?, ?, ?, ?, ?)";

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

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

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



```

        Film film = toFilm(rs);
        films.add(film);
    }
} catch (Exception ex) {
    throw new FilmException(ex.getMessage());
}
return films;
}

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

```

Complete the delete, selectAll and selectById methods

```

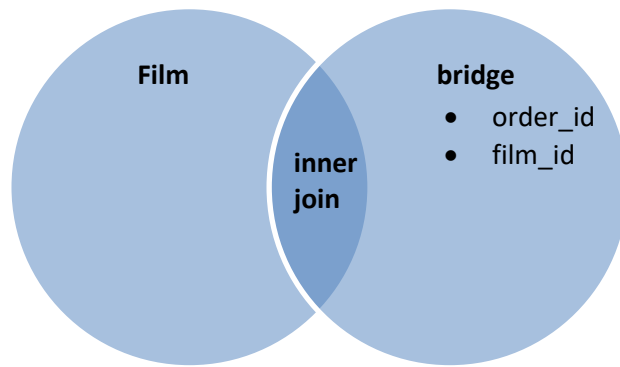
@Override
public boolean delete(Long filmId) {
    String sql = "delete from film where id = ?";

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

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

```

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.

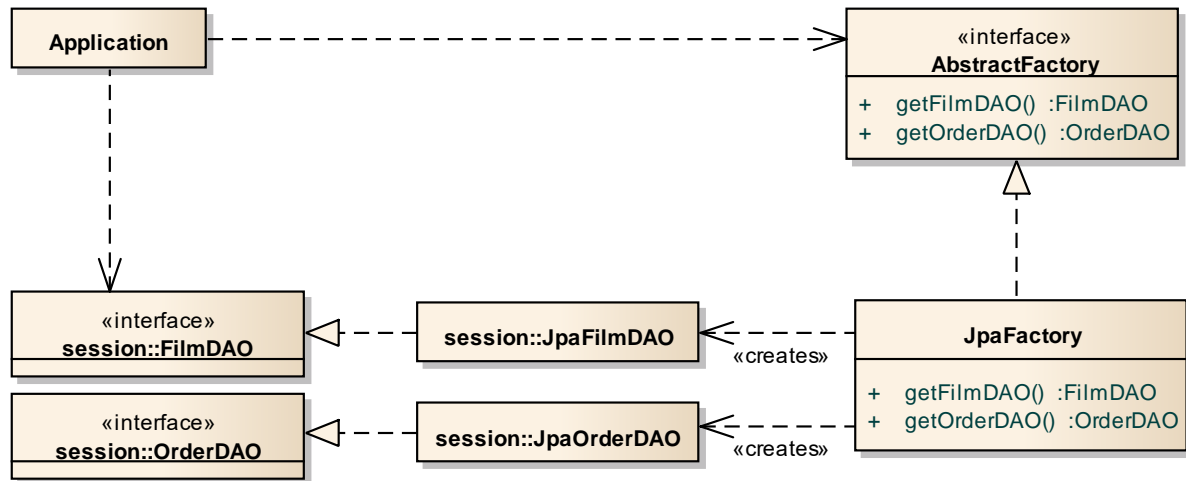
```
select title from Film
inner join bridge on Film.Id = bridge.film_id
where bridge.order_id = 1;
```

Categorising design patterns

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

Abstract Factory

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



```
public interface AbstractFactory {
    FilmDAO getFilmDAO();
    OrderDAO getOrderDAO();
}

public class JpaFactory implements AbstractFactory {
    @Override
    public FilmDAO getFilmDAO() {
        return new JpaFilmDAO();
    }
    @Override
    public OrderDAO getOrderDAO() {
        return new JpaOrderDAO();
    }
}

public class FactoryMaker {
    public static AbstractFactory getFactory(String key) {
        switch (key) {
            case "jpa":
                return new JpaFactory();
            case "jdbc":
                // return new JdbcFactory();
            default:
                return null;
        }
    }
}

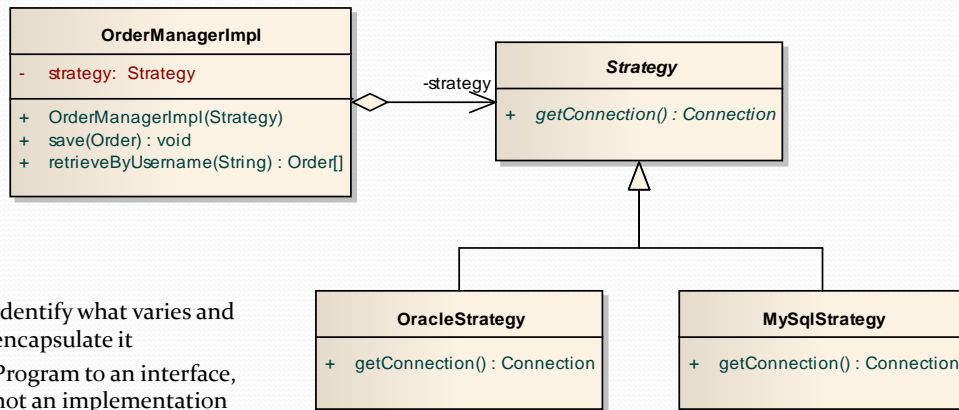
public class Application {
    public static void main(String[] args) {
        AbstractFactory af = FactoryMaker.getFactory("jpa");
        FilmDAO filmDAO = af.getFilmDAO();
        filmDAO.selectAll();
    }
}
```

```

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

```

Strategy Design Pattern

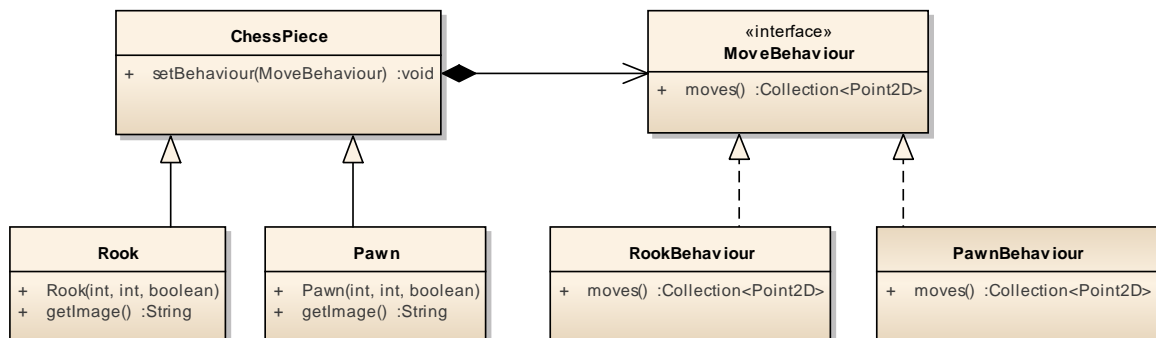


1. Identify what varies and encapsulate it
2. Program to an interface, not an implementation
3. Favour composition over inheritance

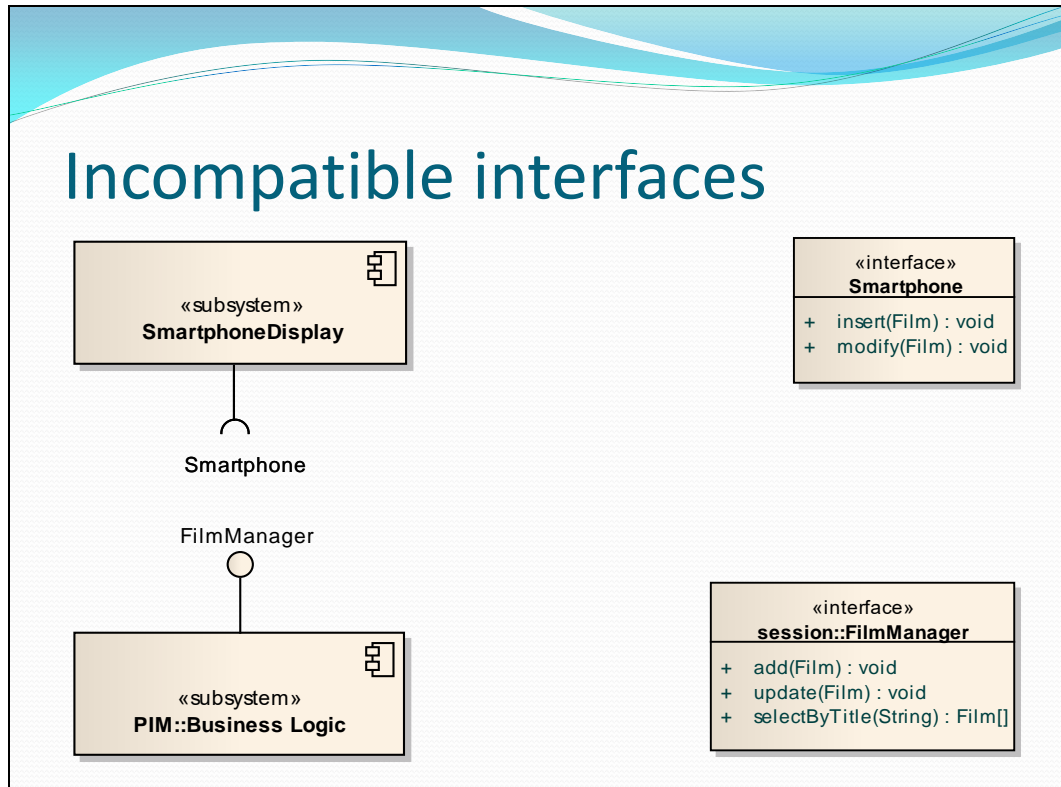
- Identify what varies and encapsulate it' so that later you can alter or extend the parts that vary without affecting those that don't. The connection behaviour of the OrderManagerImpl class is encapsulated in an abstract class called Strategy, with implementations for a number of databases.
- Program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

Using the Strategy Design Pattern

```
public class OrderManagerImpl {  
  
    private Strategy strategy;  
  
    public OrderManagerImpl(Strategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public void save(Order order) {  
        Connection c = strategy.getConnection();  
    }  
}
```

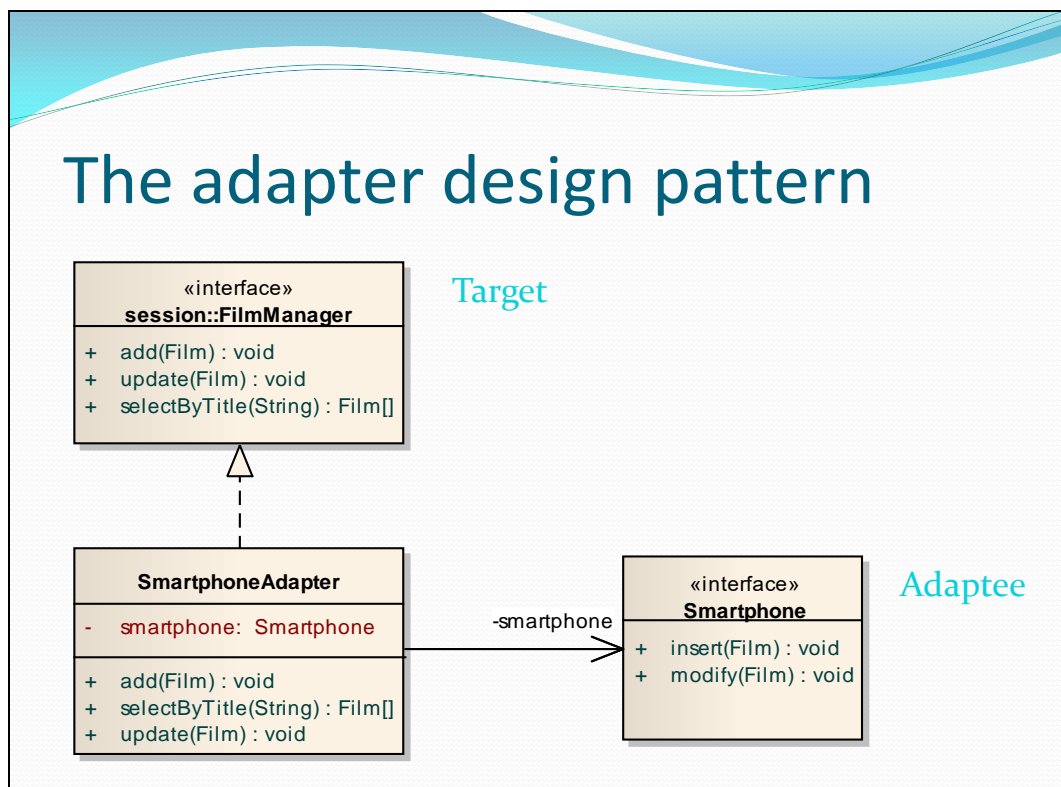


Incompatible interfaces



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

The adapter design pattern

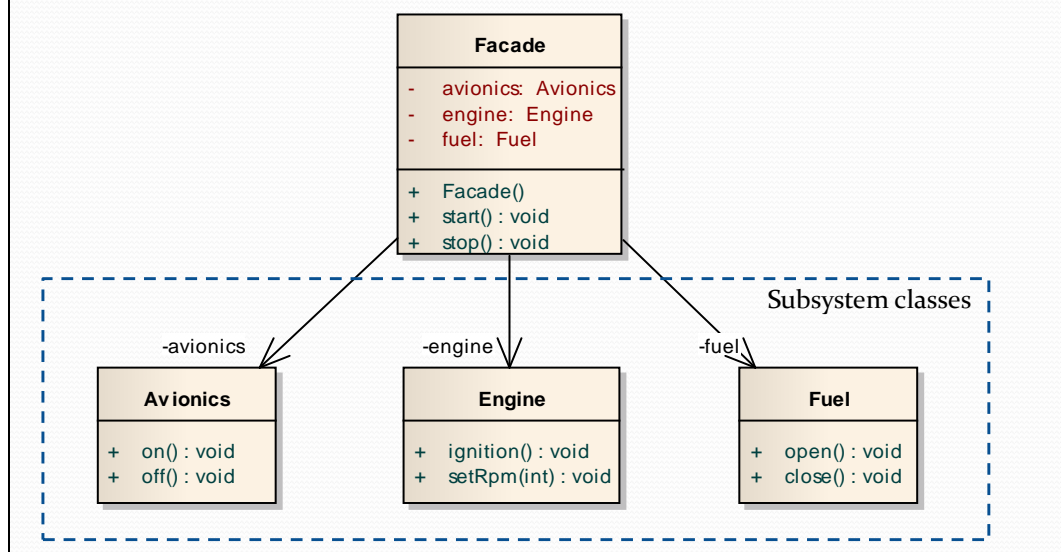


The adapter design pattern converts the interface of a class into another interface that clients expect.

The Adapter class

```
public class SmartphoneAdapter implements FileManager {  
  
    private Smartphone smartphone;  
  
    @Override  
    public void add(Film film) {  
        smartphone.insert(film);  
    }  
  
    @Override  
    public void update(Film film) {  
        smartphone.modify(film);  
    }  
}
```

The Facade Design Pattern



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

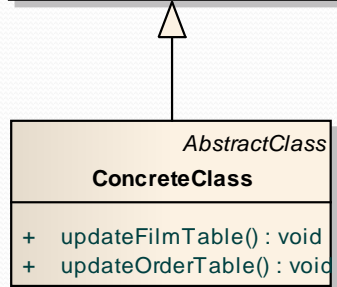
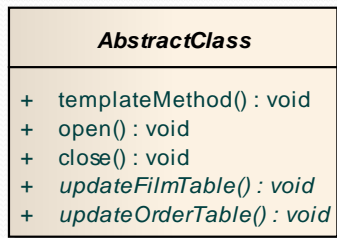
The Facade class

```
public class Facade {  
    private Avionics avionics = new Avionics();  
    private Engine engine = new Engine();  
    private Fuel fuel = new Fuel();  
  
    public void start(){  
        fuel.open();  
        engine.setRpm(1200);  
        engine.ignition();  
        avionics.on();  
    }  
  
    public void stop(){  
        engine.setRpm(1200);  
        fuel.close();  
        avionics.off();  
    }  
}
```

The above façade simplifies interaction with the subsystem.

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

Template Method Design Pattern



```

public abstract class AbstractClass {

    public void templateMethod() {
        open();
        updateFilmTable();
        updateOrderTable();
        close();
    }

    public void open() {
    }

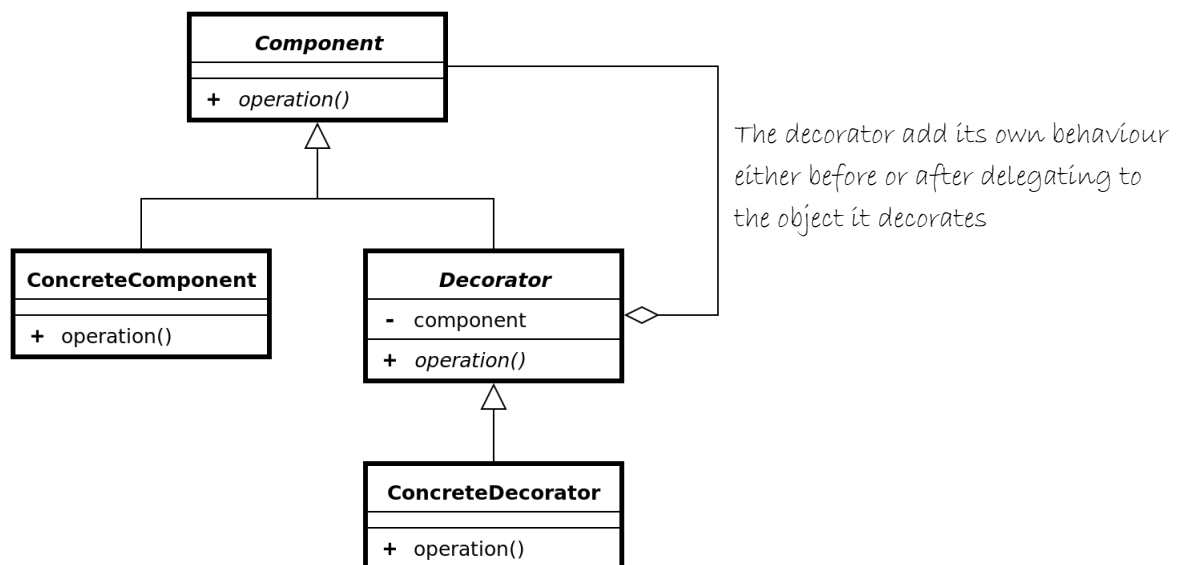
    private void close() {
    }

    public abstract void updateFilmTable();
    public abstract void updateOrderTable();
}
    
```

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

Decorator pattern

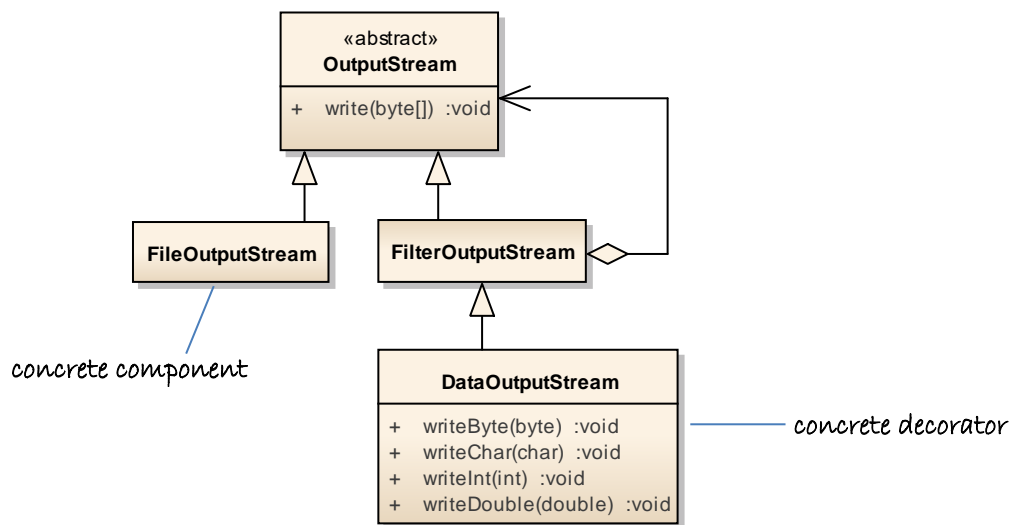
The decorator pattern attaches additional responsibilities to an object dynamically.



OO Design Principles

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

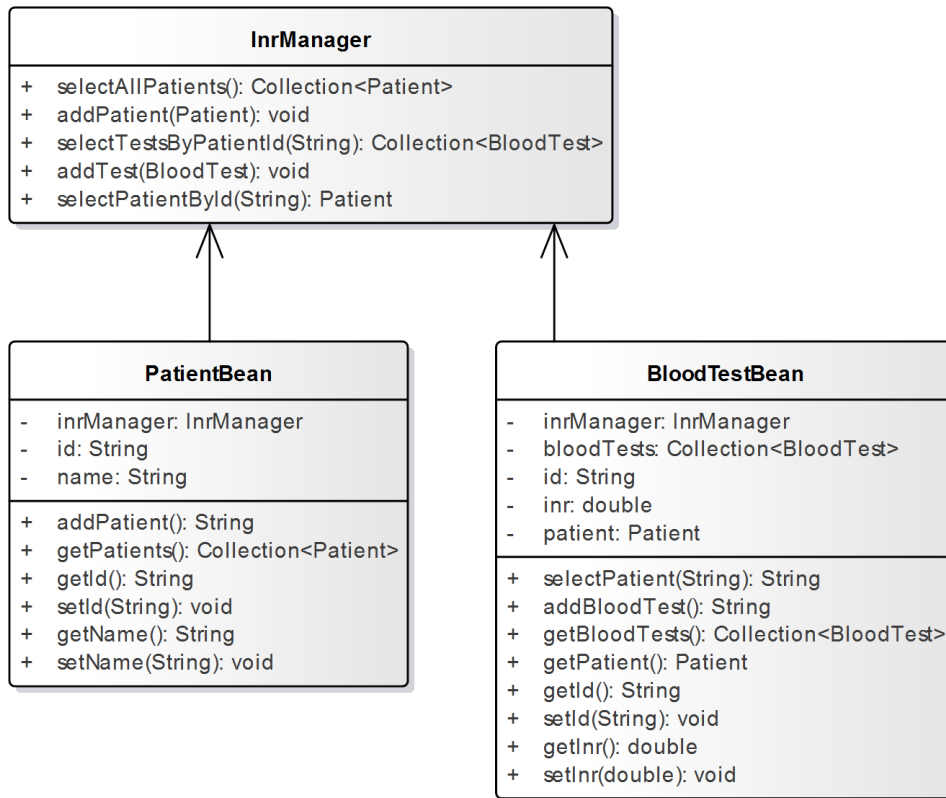
FileOutputStream



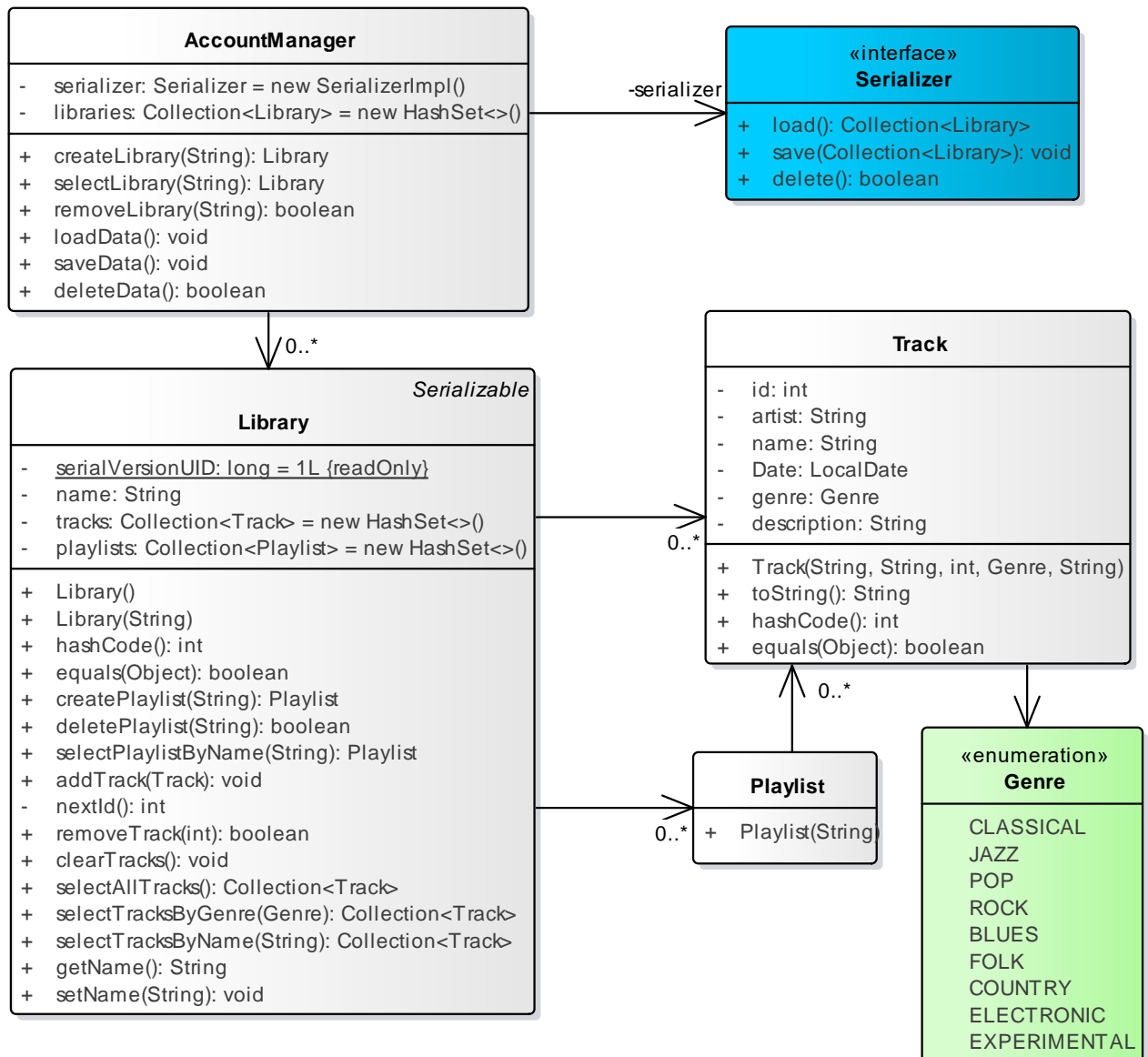
Projects

INR

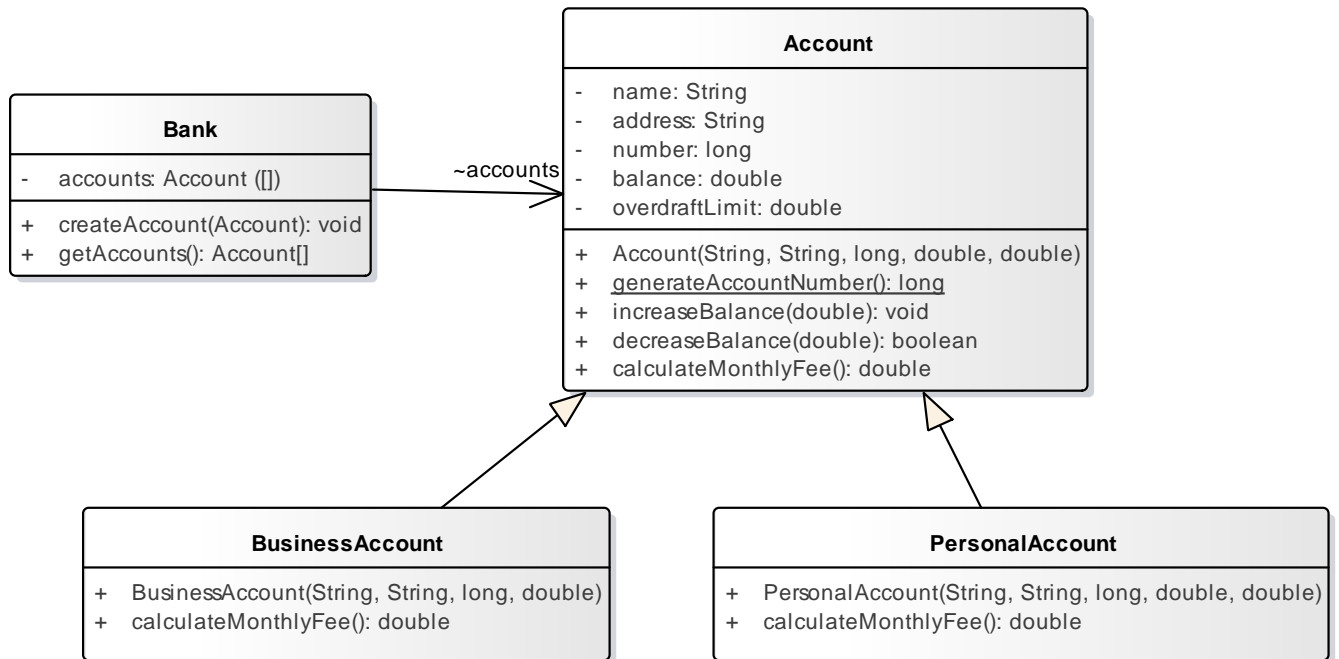
INR is based on the ratio of a patient's blood clotting time to average clotting time. Normal values are 0.8 to 1.1, or for patients on blood thinning medication, 2.0 to 3.0.



Tracks



Bank



Sample exam questions

[Exam 1 syllabus](#)

[Exam 2 syllabus](#)