Tentamen för TDA551
Objekt-orienterad Programmering och Design

Dag: 2017-01-10, *Tid:* 14.00-18.00

Ansvarig:	Alex Gerdes
Examinator:	Niklas Broberg
Förfrågningar:	Alex Gerdes (alexg@chalmers.se, 0729744966) eller Fredrik Sjöholm (0768379794)
Resultat:	Erhålls via Ladok
Betygsgränser:	3:a 24 poäng 4:a 36 poäng 5:a 48 poäng max 60 poäng
Siffror inom parentes:	Anger maximal poäng på uppgiften
Granskning:	Anslås på kurshemsidan. Vid eventuella åsikter om rättningen ange noggrant vad du anser är fel.
Hjälpmedel:	Inga (en UML cheat sheet är bifogat)
Var vänlig och:	Skriv tydligt och disponera papperet på lämpligt sätt. Börja varje uppgift på nytt blad. Skriv ej på baksidan av papperet.
Observera:	För full poäng krävs ett läsbart, begripligt och *heltäckande* svar. Svar kan ges på antingen svenska eller engelska (eller blandat). Skriv tydligt om du gör antaganden som inte står i uppgiftsbeskrivningen. Tips: Läs igenom alla frågor innan du börjar svara på uppgifter. Notera oklarheter så du kan fråga när någon kommer förbi (ca 15:00). De flesta uppgifter är designade för att ni ska ge korta svar. Få svar, om något, bör kräva mer än ett A4-ark.

Lycka till!

Börja med att betrakta och bekanta dig med en bifogade koden för "AircraftFighters" (sammanlagt 11 klasser och interfaces). De flesta av uppgifterna baseras på olika delar av denna kodbas.

(Koden innehåller användning av en viss standard-klass - närmare bestämt `Optional` - men denna har ingen betydelse för svaret på någon av frågorna. Den är bara inkluderat för att visa på ett alternativt sätt att hantera exceptionella resultat. Vi har ändå bifogat APIet längst ner.)

Uppgift 1

(4 poäng)

Redogör för följande tekniska termer, med en eller ett par meningar per svar:

a) Static method (statisk metod)

b) Dynamisk typ (dynamic type)

c) Exception (undantag)

d) Typ-parameter (type parameter)

Uppgift 2

(6 poäng)

Med utgångspunkt i koden för AircraftFighters, ange för varje deluppgift nedan om kodstycket är korrekt, ger ett statiskt fel, eller ger ett runtime-fel. Om det är korrekt, ange för varje metodanrop i kodstycket vilken eller vilka versioner av metoden `shoot` (klass och signatur) som kommer att exekveras. Om det är inkorrekt, förklara varför.

a)
```
AircraftFighter aircraft = new AircraftFighter();
aircraft.shoot();
```

b)
```
AircraftFighter aircraft = new JetFighter();
aircraft.shoot();
```

c)
```
JetFighter jetFighter = new JetFighter(new Point(0,0), 1, 100);
AircraftFighter turboJet = new JetWithTurbo(jetFighter);
turboJet.shoot();
```

d)
```
JetFighter jetFighter = new JetFighter(new Point(0,0), 1, 100);
Gun gun = GunMaker.makeStandardGun();
AircraftFighter armedJet = new JetWithGun(gun, jetFighter);
AircraftFighter turboJet = new JetWithTurbo(armedJet);
turboJet.shoot();
```

Uppgift 3

Antag att vi har följande variabler:

```
List<? extends AircraftFighter> listExtendsAF = ...;
List<AircraftFighter> listAF = ...;
List<JetFighter> listJF = ...;
List<? super JetFighter> listSuperJF = ...;
```

a) Ange för varje deluppgift nedan om kodstycket är korrekt, ger ett statiskt typfel, eller ger ett runtime-fel. Om det ger ett fel, förklara varför. Antag att alla listor innehåller minst ett element.

i.
```
listExtendsAF      = listAF;
AircraftFighter af = listExtendsAF.get(0);
```

ii.
```
listExtendsAF      = listJF;
JetFighter jf      = listExtendsAF.get(0);
```

iii.
```
listAF             = listJF;
AircraftFighter af = listAF.get(0);
```

iv.
```
listSuperJF        = listAF;
AircraftFighter af = listSuperJF.get(0);
```

(4 poäng)

b) Antag att vi vill ha en simulerings-motor för våra AircraftFighters, och att denna ska hålla reda på alla stridsflygplan i världen, oavsett hur de är sammansatta. Vilken av de fyra ovanstående listorna bör vi då välja för att lagra dessa stridsflygplan? (2 poäng)

c) Typen `List<? extends T>` är co-variant i typen `T`. Förklara vad det innebär, och visa med ett exempel vad det får för konsekvenser. (2 poäng)

Uppgift 4

a) Förklara kort vad trådsäkerhet betyder och ge (minst) ett problem som kan uppstå i samband med en brist på trådsäkerhet. (2 poäng)

b) Är klassen `CooldownGun` trådsäker? Förklara ditt svar. (2 poäng)

Uppgift 5

Betrakta klassen `Missile`:

a) Är `Missile` muterbar? Om inte, vad skulle krävas för att den ska bli det? (3 poäng)

b) Alla stridsflygplan har en position som representeras av instanser av klassen `Point`. Denna klass är muterbar, vilket kan leda till problem. De problemen kan åtgärdas vid varje användning av en `Point`, men ett bättre alternativ i det här fallet är att göra `Point` icke-muterbar. Vi vill dock fortfarande kunna ange nya `x`- och `y`-värden utifrån en gammal `Point`. Visa en alternativ implementation av `Point`, som inte är muterbar, och som använder tekniken mutate-by-copy för att tillhandahålla icke-muterande versioner av `setX`, `setY` och `move`. (3 poäng)

Uppgift 6

Redogör för följande design-principer, och ge ett eget konkret exempel för var och en av dem på kod som följer respektive inte följer principen.

a) Single Responsibility Principle (3 poäng)

b) Law of Demeter ("Don't talk to strangers") (3 poäng)

Uppgift 7

Redogör för följande design patterns (design-mönster), inklusive syfte och effekt. Visa exempel som UML-diagram:

a) Template Method Pattern (4 poäng)

b) Model-View-Controller (4 poäng)

Uppgift 8 (8 poäng)

Betrakta kodbasen för AircraftFighters. Notera alla design patterns som använts. För varje pattern, peka konkret i koden på de klasser, interfaces och metoder som bidrar. Visa gärna med hjälp av UML-diagram för relevanta delar.

Uppgift 9

Klassen `CooldownGun` har två olika tillstånd (när den kan skjuta, och när den inte kan det) vilket i nuläget specificeras av ett attribut `isInCooldown`.

a) Applicera *State Pattern* för att förbättra koden, och särskilja beteendet för dessa två tillstånd. Visa resultatet som UML-diagram, samt koden som implementerar de två olika tillstånden, samt koden som byter tillstånd. (4 poäng)

b) Applicera *Singleton Pattern* för att se till att det bara finns en instans av objektet som representerar vardera tillståndet. Visa resultatet som UML-diagram. (2 poäng)

Uppgift 10

Betrakta kodbasen för AircraftFighters. För följande design-principer, hitta minst ett fall där koden bryter mot principen:

a) Dependency Inversion Principle (2 poäng)

b) Command-Query Separation Principle (2 poäng)

AircraftFighters kodbas

AircraftFighter

```
1  import java.util.*;
2
3  public abstract class AircraftFighter {
4    private int speed = 0;
5    protected double direction = 0;
6
7    public abstract void shoot ();
8
9    public double turnLeft() {
10     direction -= 0.1;
11     return direction;
12   }
13
14   public double turnRight() {
15     direction += 0.1;
16     return direction;
17   }
18
19   public final double getDirection() { return direction; }
20
21   public final int getSpeed() { return speed; }
22
23   public abstract Point getPosition();
24   public abstract void setPosition(Point position);
25
26   public final int accelerate() {
27     setSpeed(Math.max(getTopSpeed(), getSpeed() + getAccelerationFactor()));
28     return this.speed;
29   }
30
31   public final int brake() {
32     setSpeed(Math.min(0, getSpeed() - getAccelerationFactor()));
33     return this.speed;
34   }
35
36   private void setSpeed(int speed) {
37     this.speed = speed;
38   }
39
40   protected abstract int getAccelerationFactor();
41   protected abstract int getTopSpeed();
42
43   private ArrayList<MissileListener> missileListeners = new ArrayList<>();
44
45   public final void addMissileListener(MissileListener ml) {
46     missileListeners.add(ml);
47   }
48
49   protected final void notifyListeners(Missile m) {
50     for (MissileListener ml : missileListeners)
51       ml.onMissileLaunch(m);
52   }
53 }
```

MissileListener

```java
public interface MissileListener {
  void onMissileLaunch(Missile m);
}
```

Missile

```java
final class Missile {
  private final int speed;
  private final double direction;
  private final int firepower;

  public Missile(int speed, double direction, int firepower) {
    this.speed = speed;
    this.direction = direction;
    this.firepower = firepower;
  }

  public int getSpeed() {
    return speed;
  }

  public double getDirection() {
    return direction;
  }

  public int getFirepower() {
    return firepower;
  }
}
```

Gun

```java
import java.util.Optional;

public interface Gun {
  Optional<Missile> fireMissile(Point start, double direction);
}
```

JetFighter

```
public class JetFighter extends AircraftFighter {
  private int accelerationFactor;
  private int topSpeed;
  private Point position;

  public final Point getPosition() {
    return position;
  }

  public final void setPosition(Point position) {
    this.position = position;
  }

  public JetFighter(Point position) {
    this(position, 1, 80);
  }

  public JetFighter(Point position, int accelerationFactor, int topSpeed) {
    this.position = position;
    this.accelerationFactor = accelerationFactor;
    this.topSpeed = topSpeed;
  }

  @Override
  public void shoot() {
    // do nothing, we have no gun!
  }

  @Override
  protected int getAccelerationFactor() {
    return this.accelerationFactor;
  }

  @Override
  protected int getTopSpeed() {
    return this.topSpeed;
  }
}
```

JetWithGun

```java
import java.util.Optional;

public class JetWithGun extends AircraftFighter {
  private Gun gun;
  private AircraftFighter aircraft;

  public JetWithGun(Gun gun, AircraftFighter aircraft) {
    this.gun = gun;
    this.aircraft = aircraft;
  }

  @Override
  public void shoot() {
    Optional<Missile> couldBeMissile = gun.fireMissile(this.getPosition(), this.getDirection());

    if (couldBeMissile.isPresent()) {
      notifyListeners(couldBeMissile.get());
    }
  }

  @Override
  public Point getPosition() {
    return aircraft.getPosition();
  }

  @Override
  public void setPosition(Point position) {
    aircraft.setPosition(position);
  }

  @Override
  public double turnLeft() {
    return aircraft.turnLeft();
  }

  @Override
  public double turnRight() {
    return aircraft.turnRight();
  }

  @Override
  protected int getAccelerationFactor() {
    return aircraft.getAccelerationFactor();
  }

  @Override
  protected int getTopSpeed() {
    return aircraft.getTopSpeed();
  }
}
```

CooldownGun

```
import java.util.Optional;

public class CooldownGun implements Gun {
  private boolean isInCooldown = false;
  private int cooldownLength;

  public CooldownGun(int cooldownLength) {
    this.cooldownLength = cooldownLength;
  }

  @Override
  public Optional<Missile> fireMissile(Point start, double direction) {
    if (!isInCooldown) {
      startCooldown();
      return Optional.of(new Missile(120, direction, 100));
    } else {
      return Optional.empty();
    }
  }

  private void startCooldown() {
    isInCooldown = true;
    new Cooldown().start();
  }

  class Cooldown extends Thread {
    @Override
    public void run() {
      try {
        Thread.sleep(cooldownLength);
      } catch (InterruptedException e) {
        // if interrupted, just proceed with setting variable to false.
      } finally {
        isInCooldown = false;
      }
    }
  }
}
```

GunMaker

```
public class GunMaker {
  public static Gun makeStandardGun() {
    return new StandardGun();
  }

  public static Gun makeCooldownGun() {
    return new CooldownGun(2000);
  }
}
```

JetWithTurbo

```
public class JetWithTurbo extends AircraftFighter {
  private AircraftFighter aircraft;

  public JetWithTurbo(AircraftFighter aircraft) {
    this.aircraft = aircraft;
  }

  @Override
  public void shoot() {
    aircraft.shoot();
  }

  @Override
  public Point getPosition() {
    return aircraft.getPosition();
  }

  @Override
  public void setPosition(Point position) {
    aircraft.setPosition(position);
  }

  @Override
  protected int getAccelerationFactor() {
    return aircraft.getAccelerationFactor() + 1;
  }

  @Override
  protected int getTopSpeed() {
    return aircraft.getTopSpeed() + 20;
  }

  @Override
  public double turnLeft() {
    return aircraft.turnLeft();
  }

  @Override
  public double turnRight() {
    return aircraft.turnRight();
  }
}
```

Point

```java
public class Point {
  private int x;
  private int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }

  public void move(int x, int y) {
    setX(x);
    setY(y);
  }
}
```

StandardGun

```java
import java.util.Optional;

public class StandardGun implements Gun {
  @Override
  public Optional<Missile> fireMissile(Point start, double direction) {
    return Optional.of(new Missile(200, direction, 10));
  }
}
```

UML cheat sheet

Klassdiagram: Classes

- En class består av:
  - Dess namn
  - Dess variabler
  - Dess metoder (och konstruktorer)

+ = public
- = private
= protected
~ = none (i.e. package)

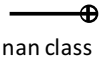
italics = abstract
underline = static
<<interface>>

Customer
+ name : String # phoneNr : PhoneNumber
+ sendOrder() : void + receiveOrder() : void

Klassdiagram: Relationer

- Fyra grundläggande typer av relationer mellan classes och interface:
  - Association (Has-A) 
    - En class har attribut (fields) som håller objekt av en annan class (interface).
  - Beroende (usage dependency) 
    - En class (interface) använder eller skapar objekt av en annan class (interface).
  - Generalisering (Is-A) 
    - En class (interface) är en direkt subclass (subinterface) till en annan class (interface).
  - Realisering (implements) 
    - En class implementerar ett interface.

Klassdiagram: Relationer specialfall

- Specialfall av association:
  - Aggregation (parts-of-a-whole) 
    - En class består av en eller flera delar, som representeras av en annan class.
  - Komposition (subpart-of) 
    - En starkare form av aggregation, där ”delen” bara får förekomma i en ”helhet”, och existerar endast så länge dess helhet existerar
  - Inkapsling (encapsulation) 
    - En class existerar inuti en annan class

`Optional` API documentation

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Class Optional<T>

java.lang.Object
    java.util.Optional<T>

```
public final class Optional<T>
extends Object
```

A container object which may or may not contain a non-null value. If a value is present, `isPresent()` will return `true` and `get()` will return the value.

Additional methods that depend on the presence or absence of a contained value are provided, such as `orElse()` (return a default value if value not present) and `ifPresent()` (execute a block of code if the value is present).

This is a value-based class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of `Optional` may have unpredictable results and should be avoided.

Since:

1.8

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`static <T> Optional<T>`	**`empty()`** Returns an empty `Optional` instance.
`boolean`	**`equals(Object obj)`** Indicates whether some other object is "equal to" this Optional.
`Optional<T>`	**`filter(Predicate<? super T> predicate)`** If a value is present, and the value matches the given predicate, return an `Optional` describing the value, otherwise return an empty `Optional`.
`<U> Optional<U>`	**`flatMap(Function<? super T,Optional<U>> mapper)`** If a value is present, apply the provided `Optional`-bearing mapping function to it, return that result, otherwise return an empty `Optional`.
`T`	**`get()`** If a value is present in this `Optional`, returns the value, otherwise throws `NoSuchElementException`.
`int`	**`hashCode()`** Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.
`void`	**`ifPresent(Consumer<? super T> consumer)`** If a value is present, invoke the specified consumer with the value, otherwise do nothing.
`boolean`	**`isPresent()`** Return `true` if there is a value present, otherwise `false`.
`<U> Optional<U>`	**`map(Function<? super T,? extends U> mapper)`** If a value is present, apply the provided mapping function to it, and if the result is non-null, return an `Optional` describing the result.
`static <T> Optional<T>`	**`of(T value)`** Returns an `Optional` with the specified present non-null value.
`static <T> Optional<T>`	**`ofNullable(T value)`** Returns an `Optional` describing the specified value, if non-null, otherwise returns an empty `Optional`.
`T`	**`orElse(T other)`** Return the value if present, otherwise return `other`.
`T`	**`orElseGet(Supplier<? extends T> other)`** Return the value if present, otherwise invoke `other` and return the result of that invocation.
`<X extends Throwable> T`	**`orElseThrow(Supplier<? extends X> exceptionSupplier)`** Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.
`String`	**`toString()`** Returns a non-empty string representation of this Optional suitable for debugging.

Methods inherited from class java.lang.Object

`clone, finalize, getClass, notify, notifyAll, wait, wait, wait`

Method Detail

empty

```
public static <T> Optional<T> empty()
```

Returns an empty `Optional` instance. No value is present for this Optional.

`Optional` API documentation

API Note:

Though it may be tempting to do so, avoid testing if an object is empty by comparing with == against instances returned by Option.empty(). There is no guarantee that it is a singleton. Instead, use isPresent().

Type Parameters:

T - Type of the non-existent value

Returns:

an empty Optional

of

`public static <T> Optional<T> of(T value)`

Returns an Optional with the specified present non-null value.

Type Parameters:

T - the class of the value

Parameters:

value - the value to be present, which must be non-null

Returns:

an Optional with the value present

Throws:

NullPointerException - if value is null

ofNullable

`public static <T> Optional<T> ofNullable(T value)`

Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.

Type Parameters:

T - the class of the value

Parameters:

value - the possibly-null value to describe

Returns:

an Optional with a present value if the specified value is non-null, otherwise an empty Optional

get

`public T get()`

If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

Returns:

the non-null value held by this Optional

Throws:

NoSuchElementException - if there is no value present

See Also:

isPresent()

isPresent

`public boolean isPresent()`

Return true if there is a value present, otherwise false.

Returns:

true if there is a value present, otherwise false

ifPresent

`public void ifPresent(Consumer<? super T> consumer)`

If a value is present, invoke the specified consumer with the value, otherwise do nothing.

Parameters:

consumer - block to be executed if a value is present

Throws:

NullPointerException - if value is present and consumer is null

filter

`public Optional<T> filter(Predicate<? super T> predicate)`

If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.

Parameters:

predicate - a predicate to apply to the value, if present

Returns:

an Optional describing the value of this Optional if a value is present and the value matches the given predicate, otherwise an empty Optional

Throws:

NullPointerException - if the predicate is null

map

`public <U> Optional<U> map(Function<? super T,? extends U> mapper)`

If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. Otherwise return an empty Optional.

`Optional` API documentation

API Note:

This method supports post-processing on optional values, without the need to explicitly check for a return status. For example, the following code traverses a stream of file names, selects one that has not yet been processed, and then opens that file, returning an Optional<FileInputStream>:

```
Optional<FileInputStream> fis =
    names.stream().filter(name -> !isProcessedYet(name))
                  .findFirst()
                  .map(name -> new FileInputStream(name));
```

Here, findFirst returns an Optional<String>, and then map returns an Optional<FileInputStream> for the desired file if one exists.

Type Parameters:

U - The type of the result of the mapping function

Parameters:

mapper - a mapping function to apply to the value, if present

Returns:

an Optional describing the result of applying a mapping function to the value of this Optional, if a value is present, otherwise an empty Optional

Throws:

NullPointerException - if the mapping function is null

flatMap

```
public <U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
```

If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. This method is similar to map(Function), but the provided mapper is one whose result is already an Optional, and if invoked, flatMap does not wrap it with an additional Optional.

Type Parameters:

U - The type parameter to the Optional returned by

Parameters:

mapper - a mapping function to apply to the value, if present the mapping function

Returns:

the result of applying an Optional-bearing mapping function to the value of this Optional, if a value is present, otherwise an empty Optional

Throws:

NullPointerException - if the mapping function is null or returns a null result

orElse

```
public T orElse(T other)
```

Return the value if present, otherwise return other.

Parameters:

other - the value to be returned if there is no value present, may be null

Returns:

the value, if present, otherwise other

orElseGet

```
public T orElseGet(Supplier<? extends T> other)
```

Return the value if present, otherwise invoke other and return the result of that invocation.

Parameters:

other - a Supplier whose result is returned if no value is present

Returns:

the value if present otherwise the result of other.get()

Throws:

NullPointerException - if value is not present and other is null

orElseThrow

```
public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
                                    throws X extends Throwable
```

Return the contained value, if present, otherwise throw an exception to be created by the provided supplier.

API Note:

A method reference to the exception constructor with an empty argument list can be used as the supplier. For example, IllegalStateException::new

Type Parameters:

X - Type of the exception to be thrown

Parameters:

exceptionSupplier - The supplier which will return the exception to be thrown

Returns:

the present value

Throws:

X - if there is no value present

NullPointerException - if no value is present and exceptionSupplier is null

X extends Throwable

equals

`Optional` API documentation

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this Optional. The other object is considered equal if:

- it is also an `Optional` and;
- both instances have no value present or;
- the present values are "equal to" each other via `equals()`.

Overrides:

`equals` in class `Object`

Parameters:

`obj - an object to be tested for equality`

Returns:

`{code true} if the other object is "equal to" this object otherwise false`

See Also:

`Object.hashCode()`, `HashMap`

hashCode

`public int hashCode()`

Returns the hash code value of the present value, if any, or 0 (zero) if no value is present.

Overrides:

`hashCode` in class `Object`

Returns:

`hash code value of the present value or 0 if no value is present`

See Also:

`Object.equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

toString

`public String toString()`

Returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions.

Overrides:

`toString` in class `Object`

Implementation Requirements:

`If a value is present the result must include its string representation in the result. Empty and present Optionals must be unambiguously differentiable.`

Returns:

`the string representation of this instance`