Java

Exception handling

Leerdoelen voor dit hoofdstuk

- Excepties in code begrijpen en identificeren.
- Bepalen hoe exceptions de normale programmastroom veranderen.
- De noodzaak begrijpen om exceptions afzonderlijk in de code af te handelen.
- Gebruik maken van *try-catch-finally* om exceptions af te handelen
- Het verschil herkennen tussen ‘checked’/’unchecked’ exceptions en errors.
- Het aanroepen van methodes die mogelijk exceptions gooien
- Het herkennen van de verschillende categorieën van excepties.

Whoops, something went wrong

Soms gaan er dingen mis tijdens de uitvoering van je programma :-(

Wat zijn fouten die jullie al zijn tegengekomen?

- Je haalt een element uit een niet-bestaande index van een array
- Je probeert iets op te halen uit een null-waarde
- Je probeert een niet-bestaand bestand te openen
- Je deelt een getal door nul
- …
- …

Klein quizje: Wat gaat er mis/kan er mis gaan?

```
public class Array {
    public static void main(String args[]) {
        String[] studenten = {"Shreya", "Joseph", null};
        System.out.println(studenten[5].length());
    }
}
```

```
public class OpenBestand {
    public static void main(String args[]) {
        FileInputStream fis = new
FileInputStream("test.txt");
    }
}
```

```
public class MethodAccess {
    public static void main(String args[])
{
        mijnMethode();
    }
    public static void mijnMethode() {
        System.out.println("mijnMethode");
        mijnMethode();
    }
}
```

Op welke regel treed een fout op? Waarom?

Waarom vangen we excepties afzonderlijk van elkaar af?

Stel je voor dat je wat opmerkingen op een blogwebsite wilt plaatsen. Om een opmerking te maken,
je moet de volgende stappen doorlopen:

1. Ga naar de blogwebsite.
2. Log in op je account.
3. Selecteer de blog waarop je wilt reageren.
4. Plaats je opmerkingen.

Ons stappenlijstje is een mooi voorbeeld van taken die allemaal afhankelijk zijn van elkaar: Je kan pas door met het volgend punt als het punt hiervoor is gelukt.

Zelf doen: Hoe zouden we deze stappen kunnen programmeren?
(Denk aan eventuele errors die zich voor kunnen doen!)

Waarom vangen we excepties afzonderlijk van elkaar af?

Als we deze stappen zouden willen programmeren creëren we een soort spaghetti:

```
// Ga naar de blogwebsite
if (websiteBeschikbaar()) {
    // Log in op je account
    if (logInSuccesvol()) {
        // Selecteer de blog waar je op wilt reageren
        if (blogBeschikbaar()) {
            // Reageer op de blog
            reageerOpBlog();
        } else {
            // database error.
        }
    } else {
        // Log in mislukt: Vraag een nieuw wachtwoord
    }
} else {
    // Website niet beschikbaar
}
```

De logica vereist dat de code de voorwaarden controleert voordat een gebruiker verder kan met de volgende stap.

Deze controle van voorwaarden op meerdere plaatsen introduceert nieuwe stappen voor gebruikers en ook nieuwe paden voor het uitvoeren van de oorspronkelijke stappen.

Het moeilijke van deze gewijzigde paden is dat ze gebruikers in verwarring kunnen brengen over de stappen die betrokken zijn bij de taken die ze proberen uit te voeren.

Kan dit netter?

Waarom vangen we excepties afzonderlijk van elkaar af?

De oplossing:

```
try {
    // Ga naar de blogwebsite
    // Log in op je account
    // Selecteer de blog waar je op wilt reageren
    // Reageer op de blog
    catch (WebsiteUnavailableException e) {}
    // defineer code voor het geval de website down is
    catch (LoginUnsuccesfulException e) {}
    // code die wordt uitgevoerd als de login mislukt is
    catch (DatabaseAccessException e) {}
    // code die wordt uitgevoerd als de database niet beschikbaar is
```

De code hiernaast definieert de oorspronkelijke stappen die nodig zijn om opmerkingen op een blog te plaatsen, samen met een code voor het afhandelen van uitzonderingen.

Omdat de exceptions afzonderlijk worden gedefinieerd, is eventuele verwarring over de stappen die je moet nemen om opmerkingen op de website te plaatsen, opgehelderd.

Bovendien doet deze code geen concessies aan het controleren van de voltooiing van een stap voordat u doorgaat naar de volgende stap, met dank aan de juiste exception handlers.

Wat gebeurt er in deze code?
Is de code leesbaarder? (waarom wel/niet?)

Oke, maar wat hebben we aan exception handling?

Afgezien van het scheiden van problemen tussen het definiëren van de normale programmalogica en de code voor het afhandelen van exceptions, kunnen exceptions ook helpen bij het vinden van de problematische code (code die een exception genereert), samen met de methode waarin deze is gedefinieerd.

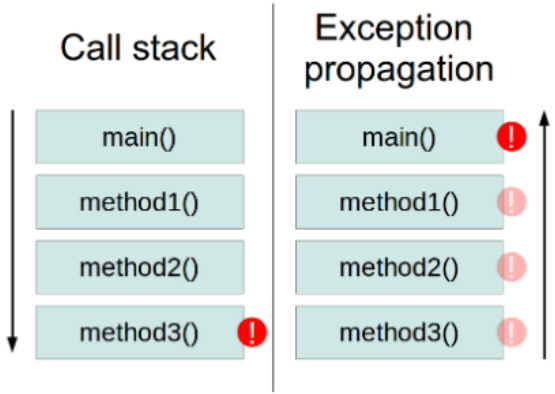
Dit doen ze door een stack trace te geven van een exception of een error.

Weet iedereen wat een stack is?

Oke, maar wat hebben we aan exception handling?

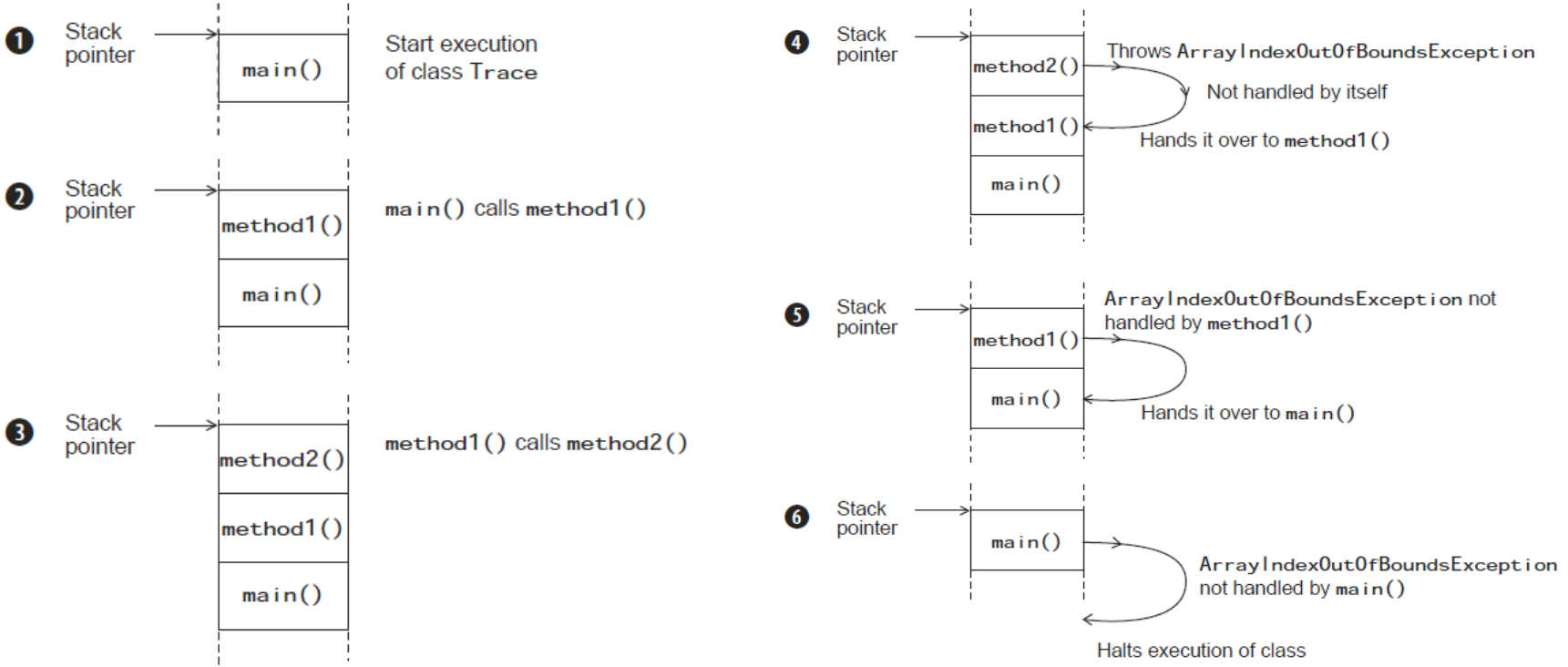
Dit doen ze door een stack trace te geven van een exception of een error.

```java
public class Trace {
    public static void methode1() {
        methode2();
    }
    public static void methode2() {
        String[] studenten = {"Shreya", "Joseph", null};
        System.out.println(studenten[5]);
    }
    public static void main(String args[]) {
        methode1();
    }
}
```

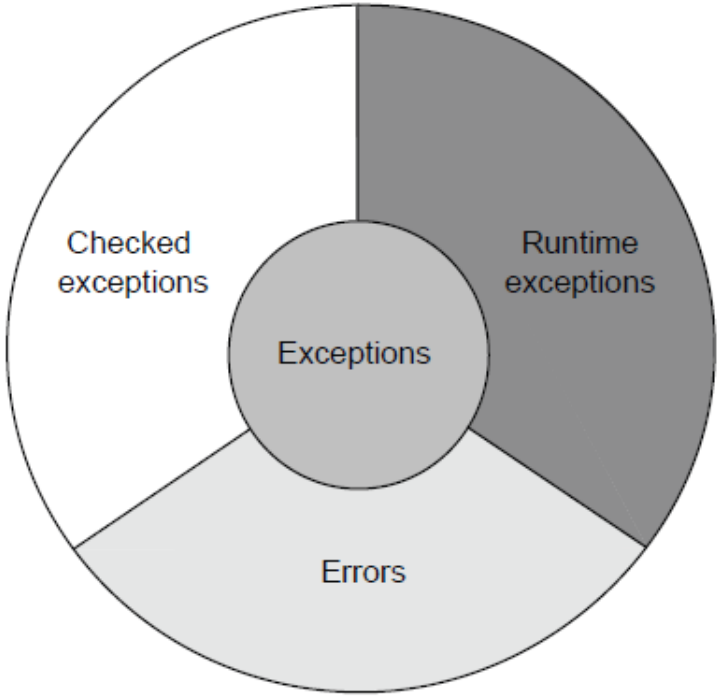



```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
at Trace.methode2(Trace.java:10)
at Trace.methode1(Trace.java:6)
at Trace.main(Trace.java:3)
```

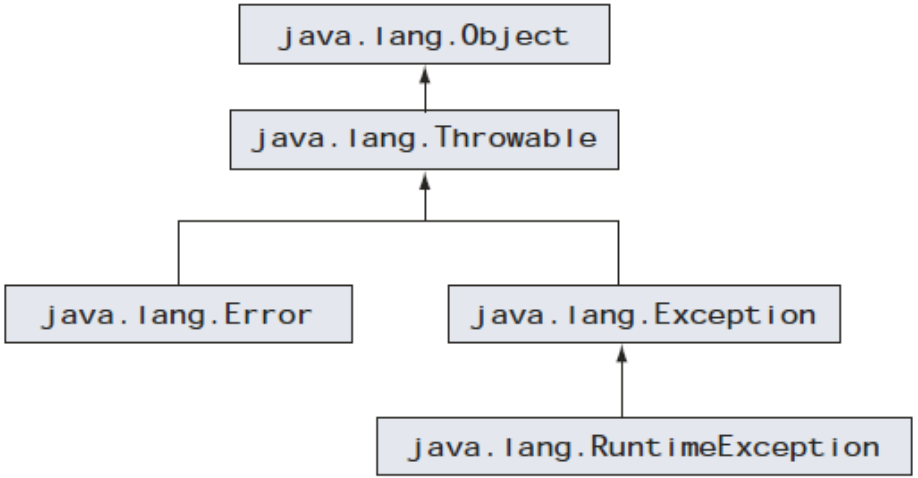
Wat gebeurt er bij een exception?



Categorieën van exceptions



Unchecked exceptions = Runtime exception + Errors



Checked exceptions

Checked exceptions zijn verreweg het meest voorkomend. Laten we daarom daar mee beginnen.

Wat denk je wat een checked exception is?

- Een checked exception is een fout die tijdens het programmeren gezien/bedacht kan worden door de programmeur.
  - Kan je een voorbeeld geven?
  - FileNotFoundException

- Checked exceptions hebben hun naam te danken aan het feit dat ze gecontroleerd worden tijdens compilatie van je code. De compiler ‘checkt’ of de programmeur wel deze mogelijke fout erkent.

Checked exceptions

Terug naar onze FileNotFound exception..

```
public class OpenBestand {
    public static void main(String args[]) {
        FileInputStream fis = new
FileInputStream("test.txt");
    }
}
```

Hoe lossen we een mogelijke error op?

```
try:
    FileInputStream fis = new FileInputStream("test.txt");
catch (FileNotFoundException e) {
    System.out.println("Bestand niet gevonden");
}
```

Programmeer deze oplossing eens en kijk of het werkt.

Runtime exceptions

Hoewel we het grootste deel van onze tijd en energie bezig zijn met checked exceptions, zullen de runtime-exceptions ons de meeste hoofdpijn bezorgen.

Wat is een Runtime exception en wat zijn voorbeelden van Runtime exceptions?

- Runtime errors ontstaan terwijl je code wordt uitgevoerd. Dit is dus niet iets waar je van te voren rekening mee kan houden. Zoals of wel of niet een bestand bestaat.
- NullPointerException, ArrayIndexOutOfBounds-excpetion, ClassCastException.

Errors

Tot nu toe hebben we nog wel het meest te maken gehad met errors. Wat zijn voorbeelden hiervan?

… Waneer je code niet compileert

Methodes die exceptions gooien

Waarom denk je dat we methoden nodig hebben die exception kunnen ‘gooien’ (throw)

Stel je voor dat je de opdracht hebt gekregen om een specifiek boek te zoeken, deze te lezen en vervolgens alles wat er in staat te vertellen. Dan kunnen dit de stappen zijn die je maakt:

1. Zoek het boek
2. Lees alle bladzijdes
3. Leg uit wat er in het boek staan aan andere mensen

Maar wat gebeurt er als we dit boek niet kunnen vinden? We kunnen dan niet meer door met de rest van onze stappen.

Als dit het geval is moet je terug naar de persoon die je de taak heeft gegeven en vertellen dat je geen boek hebt kunnen vinden. Door dit aan te geven kan deze persoon ook weer van plan wijzigen.

Methodes die exceptions gooien

Laten we ons voorbeeld eens gaan programmeren:

```
void geefLes() throws BoekNietGevondenException {
    boolean boekGevonden = zoekBoek();

    if (!boekGevonden) {
        throw new BoekNietGevondenException();
    } else {
        leesboek();
        legUit();
    }
}
```

We gebruiken hier **throw** en **throws** let goed op waar je welk keyword gebruikt.

Methodes die exceptions gooien

We breiden het voorbeeld uit door onze eigen exceptie te programmeren

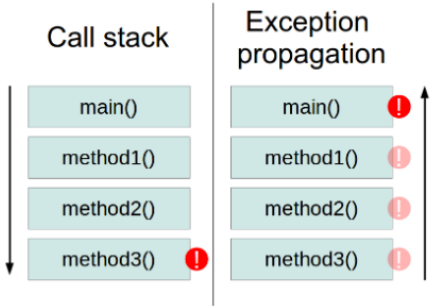
We kunnen onze eigen exceptions maken omdat excepties, net als bijna alles in Java ook classes zijn. Deze classes kunnen we extenden en door middel het maken van onze constructor onze eigen foutmelding teruggeven.

```
class BoekNietGevondenException extends Exception {
    public BoekNietGevondenException() {
        super("Boek niet gevonden");
    }
}

..

catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
```

Throw/Throws

In dit voorbeeld maken we een class *ThrowExceptions* die verschillende methoden definieert. Elke methode gooit eer een ander type exception. Compileert deze code?

```
class ThrowExceptions {
    void methode1() throws Error {}
    void methode2() throws Exception {}
    void methode3() throws Throwable {}
    void methode4() throws RuntimeException {}
    void methode5() throws FileNotFoundException {}
}
```



Ja. Had je dat verwacht? Waarom wel/niet?
Deze methodes zeggen dus dat het niet hun probleem is om een probleem op te lossen en gooien het gewoon omhoog in de stack.

Throw/Throws

Laten we de code iets uitbreiden.
Wat gebeurt er nu met onze code?

```
class ThrowExceptions {
    void methode1() throws Error {}
    void methode2() throws Exception {}
    void methode3() throws Throwable {}
    void methode4() throws RuntimeException {}
    void methode5() throws FileNotFoundException {}
    …
```

```
error: exception FileNotFoundException is never thrown in
body of corresponding try statement
        } catch (FileNotFoundException e) {
          ^
1 error
```

Compileert niet…
Checked exceptions moeten opgelost worden.

```
    void methode6() {
        try {}
        catch (Error e) {}
    }
    void methode7() {
        try {}
        catch (Exception e) {}
    }
    void methode8() {
        try {}
        catch (Throwable e) {}
    }
    void methode9() {
        try {}
        catch (RuntimeException e) {}
    }
    void method10() {
        try {}
        catch (FileNotFoundException e) {}
    }
```

Throw/Throws

```java
import java.io.FileNotFoundException;

public class Test {
    void fileNotFoundTest() throws FileNotFoundException {
        throw new FileNotFoundException();
    }

    void catchFileNotFound() {
        try {
            fileNotFoundTest();
        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        }
    }
}
```

Nu werkt onze checked exception wel…

Try-catch-finally

Wanneer je met exceptions werkt, hoor je vaak de termen **try**, **catch** en **finally**. Voordat je met deze concepten aan de slag gaat, beantwoord ik drie simpele vragen:

- Try wat?
  - Allereerst *probeer* je je code uit. Als dit fout gaat kan je dit oplossen in het *catch* blok
- Catch wat?
  - Je *vangt* het probleem dat is ontstaan in de *try* gedeelte van je code. Je lost het probleem op.
- Finally?!
  - Ten slotte voer je een set code uit, onder alle omstandigheden, ongeacht of de code in het try-blok exceptions genereert.

Dus: Try-catch-finally

We zagen eerder al de Try-Catch terug komen.
Het try-catch-statement bevat 2 verplichten onderdelen: try en catch.
(of alleen try en finally, dat werkt ook)

```
try {
    // De code hier wordt uitgevoerd totdat er iets mis gaat
} catch(NullPointerException e) {
    // Als de opgegooide exception een NullPointerException was
    // Wordt deze code uitgevoerd
} catch(Exception e) {
    // Als de exception geen NullPointerException was,
    // maar wel een Exception wordt deze code uitgevoerd
} finally {
    // Deze code wordt altijd uitgevoerd, wel of geen exception
}
```

Delen door 0

Bij het delen door 0 wordt een *ArithmeticException* gegooid.
De hierarchie-boom ziet er als volgt uit:

- java.lang.Exception
  - java.lang.RunetimeException
    - java.lang.ArithmeticException

Delen door 0

Bij het delen door 0 wordt een *ArithmeticException* gegooid.
De hierarchie-boom ziet er als volgt uit:

- java.lang.Exception
  - java.lang.RunetimeException
    - java.lang.ArithmeticException

```
try {
    int a = 3 / 0;
}
catch(ArithmeticException e) {
    System.out.println("Mag niet!");
}
```

De volgorde is essentieel

```
try {
    int a = 3 / 0;
}
catch(RuntimeException e) {
    System.out.println("Mag niet!");
}
catch(ArithmeticException e) {
    System.out.println("Mag ook niet");
}
```

Wat print deze code?

Je vangt met catch(...Exception e) alle exceptions op van dat type, dus zowel de directe instanties van dat object, als ook alle instanties van de subklassen!

Veel voorkomende erros

Runtime exceptions	Errors
`ArrayIndexOutOfBoundsException`	`ExceptionInInitializerError`
`IndexOutOfBoundsException`	`StackOverflowError`
`ClassCastException`	`NoClassDefFoundError`
`IllegalArgumentException`	`OutOfMemoryError`
`ArithmeticException`	
`NullPointerException`	
`NumberFormatException`	

ArrayIndexOutOfBounds

```
String[] seizoen = {"Lente", "Zomer", "Herfst", "Winter"};
System.out.println(seizoen[5]);
System.out.println(seizoen[-9]);
```

IndexOutOfBoundsException

```
ArrayList<String> examen = new ArrayList<>();
examen.add("SCJP");
examen.add("SCWCD");

System.out.println(examen.get(-1));
System.out.println(examen.get(4));
```

ClassCastException

```
import java.util.ArrayList;
public class ListToegang {
    public static void main(String args[]) {
        ArrayList<Inkt> inkts = new ArrayList<Inkt>();
        inkts.add(new KleurInkt());
        inkts.add(new ZwarteInkt());
        Inkt inkt = (ZwarteInkt)inkts.get(0); // gooit een ClassCastException.
    }
}

class Inkt{}
class KleurInkt extends Inkt{}
class ZwarteInkt extends Inkt{}
```

KleurInkt kan niet worden gecast naar **ZwarteInkt.**
Op te lossen met *isinstanceOf()*

IlligalArgumentException

```
public void login(String gebruikersnaam, String wachtwoord, int maxLoginProbeer) {
    if (gebruikersnaam == null || gebruikersnaam.length() < 6)
        throw new IllegalArgumentException("Login:gebruikersnaam kan niet korter zijn dan 6 karakters");
    if (wachtwoord == null || wachtwoord.length() < 8)
        throw new IllegalArgumentException("Login: wachtwoord kan niet korter zijn dan 8 karakters");
    if (maxLoginProbeer < 0)
        throw new IllegalArgumentException("Login: foute invoer waarde");
```

NullPointerException

```
import java.util.ArrayList;
class ThrowNullPointerException {
    static ArrayList<String> list = null;
    public static void main(String[] args) {
        list.add("1");
    }
}
```

ArithmeticException

```
class ThrowArithmeticEx {
    public static void main(String args[]) {
        int a = 10;
        int y = a++;
        int z = y--;
        int x1 = a - 2*y - z;
        int x2 = a - 11;
        int x = x1/ x2;
        System.out.println(x);
    }
}
```

Wat is de uitvoer van deze code?

ArithmeticException

```
public static void main(String args[]) {
        System.out.println(77.0/0);
    }
```

Wat is de uitvoer van deze code?

ArithmeticException

```
class DivideByZeroPointZero {
    public static void main(String args[]) {
        int a = 10;
        int y = a++;
        int z = y--;
        int x1 = a - 2 * y - z;
        int x2 = a - 11;
        double x3 = x2;
        double x = x1 / x3;
        System.out.println(x);
        System.out.println(x1);
        System.out.println(x3);
    }
}
```

Wat is de uitvoer van deze code?

NumberFormatException

```
System.out.println(Integer.parseInt("-123"));
System.out.println(Integer.parseInt("123"));
System.out.println(Integer.parseInt("+123"));
System.out.println(Integer.parseInt("123_45"));
System.out.println(Integer.parseInt("12ABCD"));
```

Waar is de exception?

ExceptionInInitializerError

```
public class DemoExceptionInInitializerError {
    static {
        int num = Integer.parseInt("sd", 16);
    }
}
```

Wat is de error?

StackOverflowError

```
public class DemoStackOverflowError{
    static void recursion() {
        recursion();
    }
    public static void main(String args[]) {
        recursion();
    }
}
```

Wat is de error?

Try-catch-finally

```
class ValInRivierException extends Exception {}
class LaatRoeispaanVallenException extends Exception {}
```

We maken twee exceptions

```
class RivierRaften {
    void doorStroomVersnelling(int snelheid) throws ValInRivierException
{
        System.out.println("Door stroomversnelling");
        if (snelheid > 10) throw new ValInRivierException();
    }
    void RoeiBoot(String staat) throws LaatRoeispaanVallenException {
        System.out.println("Roei boot");
        if (staat.equals("nervous")) throw new DropOarException();
    }
}
```

Een voorbeeld van het raften

Try-catch-finally

```
public static void main(String args[]) {
        RivierRaften rivierRaften = new RivierRaften();
        try {
            rivierRaften.doorStroomversnelling(11); // Als we deze <10 houden hebben we geen exceptions
            rivierRaften.roeiBoot("blij");
            System.out.println("Lekker aan het raften!")
        }
        catch (ValInRivierException e) {
            System.out.println("Ga terug de raft in!");
        }
        catch (LaatRoeispaanVallenException e) {
            System.out.println("Raak niet in paniek");
        }
        finally {
            System.out.println("Ik betaal voor de activiteit");
        }
    }
```

Eindopdracht

Laten we Try-Catch-Finally eens oefenen met een real-life voorbeeld

- Stel je voor dat je gaat raften op de rivier tijdens je vakantie.
- Je instructeur informeert je dat je tijdens het raften van het vlot in de rivier kunt vallen terwijl je de stroomversnellingen oversteekt.
- In een dergelijke toestand moet je proberen je roeispaan of het naar je toe gegooide touw te gebruiken om terug in het vlot te komen.
- Je kunt ook je roeispaan in de rivier laten vallen terwijl je op je vlot roeit. In een dergelijke toestand moet je niet in paniek raken en moet je blijven zitten.
- Wat er ook gebeurt, je moet betalen voor voor de raft.

Probeer dit voorbeeld eens te programmeren!
Tip: Gebruik niet alleen een try-catch. Maak ook methodes

Leerdoelen van dit hoofdstuk

- ✓ Excepties in code begrijpen en identificeren.
- ✓ Bepalen hoe exceptions de normale programmastroom veranderen.
- ✓ De noodzaak begrijpen om exceptions afzonderlijk in de code af te handelen.
- ✓ Gebruik maken van *try-catch-finally* om exceptions af te handelen
- ✓ Het verschil herkennen tussen ‘checked’/’unchecked’ exceptions en errors.
- ✓ Het aanroepen van methodes die mogelijk exceptions gooien
- ✓ Het herkennen van de verschillende categorieën van excepties.