

Testning och felhantering

Föreläsning 8

TDA540 - Objektorienterad Programmering



CHALMERS

Sammanfattning

- Textfiler
- Flerdimensionella fält
- ArrayList (dynamiska vs. statiska datastrukturer)
- Shorthand operatorer

Testning

Hur förvissas vi oss om att bredvidstående program är korrekt?

Genom testning!!!

Modulär design underlättar testning, eftersom varje metod kan testas individuellt.

Gör en modulär design av programmet!

```
import javax.swing.*;
public class Postage {
    public static void main( String[] arg) {
        String input = JOptionPane.showInputDialog("Ange vikten:");
        double weight = Double.parseDouble(input);
        String output;
        if (weight <= 0.0)
            output = "Du har angivit en ogiltig vikt!!!";
        else if (weight <=20.0)
            output = "Portot är 5.50 kronor.";
        else if (weight <= 100.0)
            output = "Portot är 11.00 kronor.";
        else if (weight <=250.0)
            output = "Portot är 22.00 kronor.";
        else if (weight <=500.0)
            output = "Portot är 33.00 kronor.";
        else
            output = "Måste gå som paket.";
        JOptionPane.showMessageDialog(null, output);
    } // main
} // Postage
```

Modulär design av Postage

```
import javax.swing.*;
public class Postage {
    public static void main( String[] arg) {
        String input = JOptionPane.showInputDialog("Ange vikten:");
        double weight = Double.parseDouble(input);
        JOptionPane.showMessageDialog(null, getPostage(weight));
    } // main

    public static String getPostage(double weight) {
        if (weight <= 0.0)
            return "Du har angivit en ogiltig vikt!!";
        else if (weight <=20.0)
            return "Portot är 5.50 kronor.";
        else if (weight <= 100.0)
            return "Portot är 11.00 kronor.";
        else if (weight <=250.0)
            return "Portot är 22.00 kronor.";
        else if (weight <=500.0)
            return "Portot är 33.00 kronor.";
        else
            return "Måste gå som paket.";
    } //getPostage
} // Postage
```

Fel i program

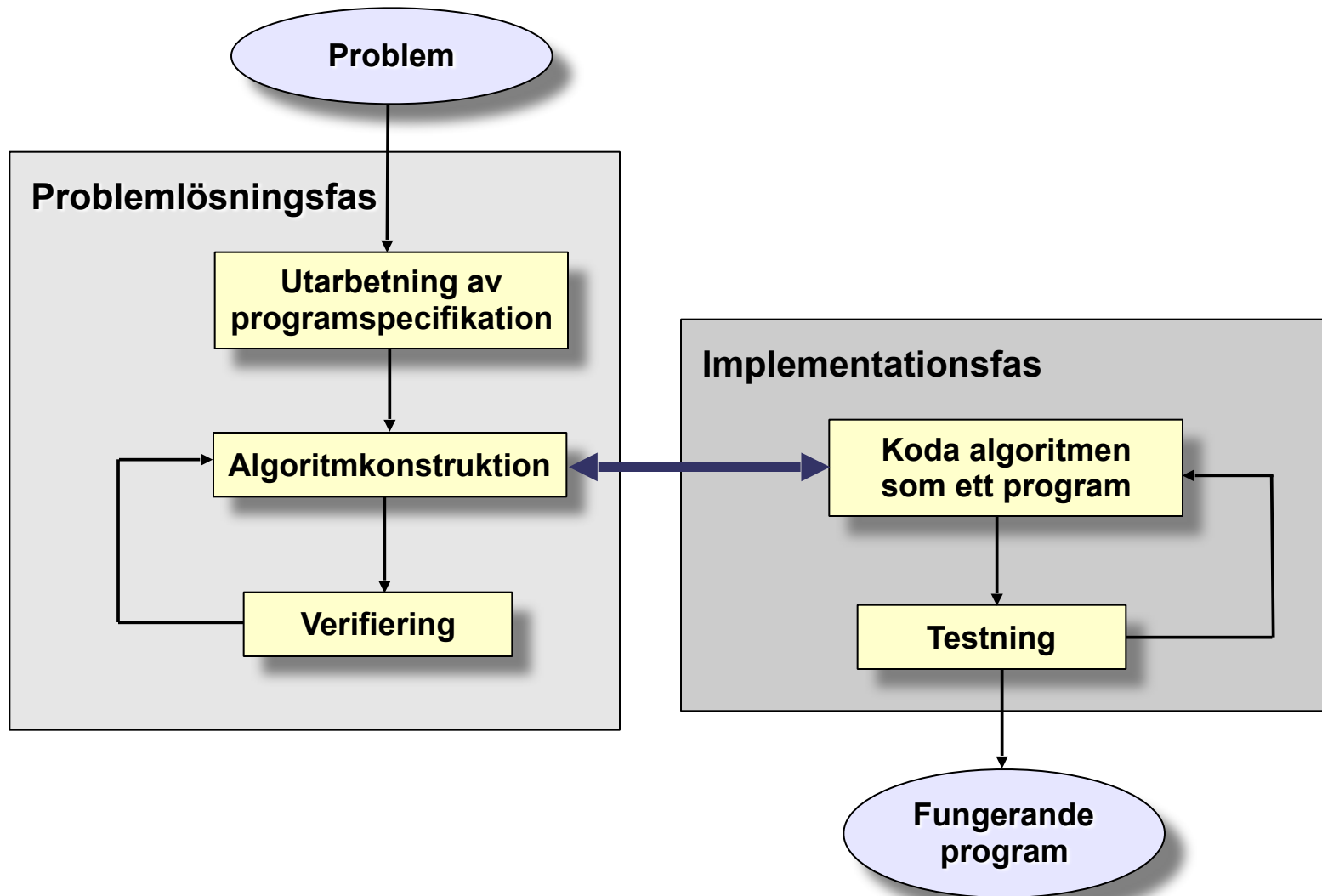
När man skriver program uppkommer alltid fel. Felen kan indelas i följande kategorier:

- Under kompileringen upptäcker kompilatorn fel som handlar om att man använt konstruktionerna i programspråket på ett felaktigt sätt (*kompileringsfel*). Kompileringsfelen kan indelas i *syntaktiska fel* och *semantiska fel*.
- I ett exekveringsbart program kan det förekomma två typer av fel - *exekveringsfel* och *logiska fel*.
 - Exekveringsfel uppkommer t.ex. på grund av att det under exekveringen någonstans i programmet sker en evaluering av ett uttryck som resulterar i att ett värde erhålls som ligger utanför det giltiga definitionsområdet för uttryckets datatyp. Felen uppträder vanligtvis inte vid varje körning utan endast då vissa specifika sekvenser av indata ges till programmet.
 - Logiska fel är rena tankefel hos programmeraren. Exekveringen lyckas, men programmet gör inte vad det borde göra.

Testning av program

- Testning är en vedertagen metod inom all ingenjörsmässig verksamhet för att fastställa om en hypotes, konstruktion eller produkt är korrekt och fungerar som avsett.
- Till grund för all testning av program ligger *programspecifikationen*. En dålig eller felaktig specifikation leder naturligtvis till ett undermåligt eller felaktigt program.
- En *testplan*, som anger hur det färdiga programmet skall fungera, bör utarbetas samtidigt med programspecifikationen.
- Testning är ett sätt att minimera antalet fel i ett program.
- Testning kan ***enbart påvisa förekomsten av fel, aldrig frånvaron av fel!***
- Testning skall ske i samtliga faser av programutvecklingen.

Verklig modell för programutveckling



Testning av program

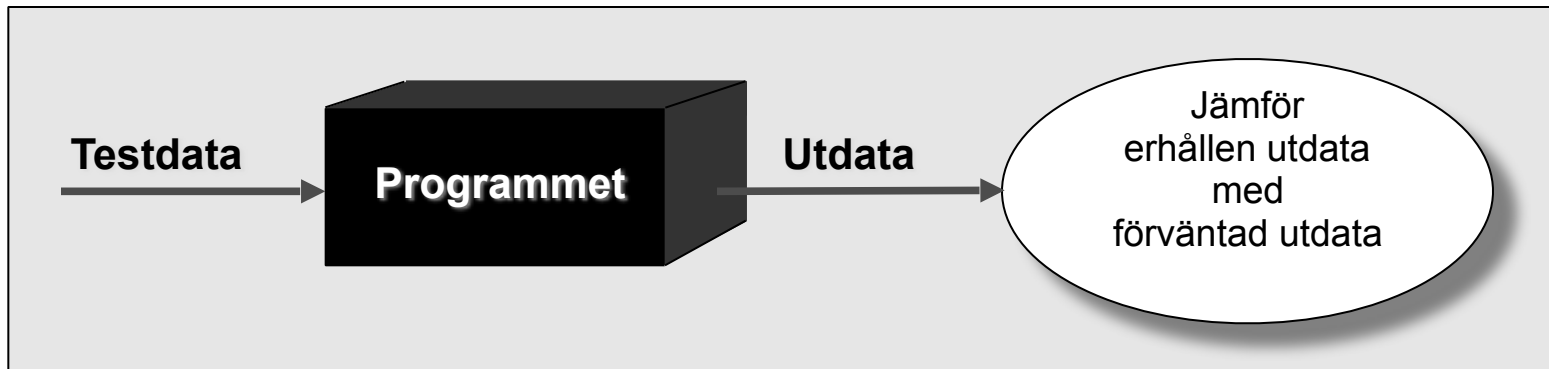
Ju senare i utvecklingsarbetet man upptäcker ett fel ju svårare och dyrare är det att lokalisera och korrigera felet.

En algoritm testas/verifieras manuellt genom att gå igenom algoritmen och utför de olika stegen i algoritmen. Syftet är att verifiera att algoritmen inte innehåller några *logiska fel*.

Under testning av algoritmen kan det visa sig att programspecifikationen är felaktig eller ofullständig, då måste man backa tillbaks till programspecifikationen och göra nödvändiga modifieringar eller kompletteringar.

Testning av program

Vid *leveranstestning* görs *black-box testning*, vilket innebär att testningen sätts upp utan kunskaper om hur programmet är implementerat.



Om felaktigheter upptäcks under testningen, skall felen naturligtvis åtgärdas. Därefter skall alla testfallen i testplanen återupprepas, eftersom korrigeringar av fel kan introducera nya fel.

Testning av program

Det är omöjligt att göra *uttömmande testning*, dvs testa alla uppsättningar av möjliga indatasekvenser.

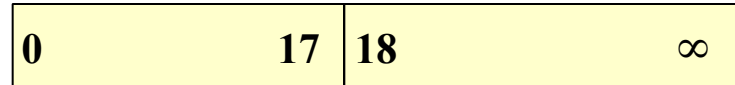
Men vi måste ha en uppsättning testfall som är så övertygande att man kan *anta* att programmet är korrekt.

Den metod som används är att indela de möjliga indatasekvenserna i olika *ekvivalens kategorier*.

Testning av program

Exempel:

Anta att vi har ett program som skall skriva ut om en person får rösta eller inte. Röståldern är 18 år. Vi får då två ekvivalens kategorier enligt nedanstående figur:



Det finns dock ytterligare en ekvivalens kategorier, nämligen den som består av ogiltig data.

I testplanen väljs (minst) ett testfall från varje ekvivalens kategorier, samt testfall med värden från övergångarna mellan ekvivalens kategorierna. Vi får således följande testplan:

<u>Test nr</u>	<u>Indata</u>	<u>Förväntat resultat</u>
1	12	Får inte rösta
2	21	Får rösta
3	17	Får inte rösta
4	18	Får rösta
5	-10	Felutskrift
6	0	Får inte rösta
7	-1	Felutskrift

Testprogram för metoden `getPostage`:

```
public class TestGetPostage {
    public static void main( String[] arg) {
        System.out.print("Testfall 1: Förväntat värde: Du har angivit en ogiltig vikt!! ");
        System.out.println("Erhållet värde: " + Postage.getPostage(0));
        System.out.print("Testfall 2: Förväntat värde: Portot är 5.50 kronor.");
        System.out.println("Erhållet värde: " + Postage.getPostage(0.5));
        System.out.print("Testfall 3: Förväntat värde: Portot är 5.50 kronor.");
        System.out.println("Erhållet värde: " + Postage.getPostage(20.0));
        System.out.print("Testfall 4: Förväntat värde: Portot är 11.00 kronor.");
        System.out.println("Erhållet värde: " + Postage.getPostage(20.5));
        System.out.print("Testfall 5: Förväntat värde: Portot är 11.00 kronor.");
        System.out.println("Erhållet värde: " + Postage.getPostage(100.0));
        System.out.print("Testfall 6: Förväntat värde: Portot är 22.00 kronor.");
        System.out.println("Erhållet värde: " + Postage.getPostage(100.5));
        System.out.print("Testfall 7: Förväntat värde: Portot är 22.00 kronor.");
        System.out.println("Erhållet värde: " + Postage.getPostage(250.0));
        System.out.print("Testfall 8: Förväntat värde: Portot är 33.00 kronor. ");
        System.out.println("Erhållet värde: " + Postage.getPostage(250.5));
        System.out.print("Testfall 9: Förväntat värde: Portot är 33.00 kronor.");
        System.out.println("Erhållet värde: " + Postage.getPostage(500.0));
        System.out.print("Testfall 10: Förväntat värde: Måste gå som paket.");
        System.out.println("Erhållet värde: " + Postage.getPostage(500.5));
    } //main
} //TestGetPostage
```

Testprogram för metoden `getPostage`:

```
public class TestPostage {
    static int nr = 1;

    public static void main( String[] arg) {
        boolean res =
            testPostage(0,      "Du har angivit en ogiltig vikt!!") &&
            testPostage(0.5,   "Portot är 5.50 kronor.") &&
            testPostage(20.0,  "Portot är 5.50 kronor.") &&
            testPostage(20.5,  "Portot är 11.00 kronor.") &&
            testPostage(100.0, "Portot är 11.00 kronor.") &&
            testPostage(100.5, "Portot är 22.00 kronor.") &&
            testPostage(250.0, "Portot är 22.00 kronor.") &&
            testPostage(250.5, "Portot är 33.00 kronor.") &&
            testPostage(500.0, "Portot är 33.00 kronor.") &&
            testPostage(500.5, "Måste gå som paket.");

        String out = res ? "All tests pass!" : "A test failed!";
        System.out.println("\n" + out);
    } //main

    public static boolean testPostage(double weight, String expected) {
        String result = Postage.getPostage(weight);

        System.out.printf("Testfall %d: ", nr++);
        System.out.print("Förväntat värde: " + expected + " ");
        System.out.println("Erhållet värde: " + result);

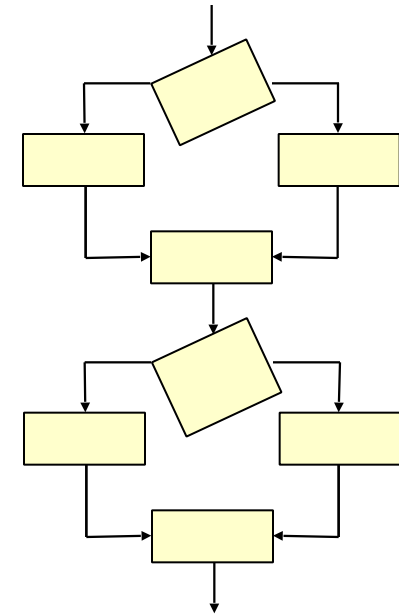
        return result.equals(expected);
    }
} //TestGetPostage
```

White-box-testning

När man testar sina programkomponenter under implementationsfasen har man kännedom om hur implementationen är gjord.

Testmetoder som drar nytta av att implementationen är tillgänglig kallas för *white-box testning*.

Vid white-box testning används kunskapen om programmets strukturella uppbyggnad när testdata väljs.



Det man eftersträvar vid white-box testning är att köra programmet med en uppsättning testfall som valts på så sätt att varje sats i programmet blir exekverad under testningen.

I stora program finns det enormt många olika exekveringsvägar, varför det i praktiken är omöjligt att göra en fullständig white-box testning.

Testning av komponenter och system

Varje programenhet skall testas separat från övriga enheter innan enheten integreras i programsystemet. Man då har större möjlighet att lokalisera och åtgärda uppkomna fel.

Att testa en programkomponent kallas för *enhetstesting*.

Att testa det kompletta programsystemet kallas för *systemtesting*.

Det finns två olika metoder för att testa de enskilda enheterna i ett programsystem, *bottom-up* och *top-down*.

Vid *bottom-up* utvecklas och testas de minsta och mest grundläggande enheterna först, varefter dessa kan användas som komponenter i större och mera kraftfulla enheter.

Vid *top-down* utvecklas och testas de största och mest abstrakta enheterna först. Då *top-down* är en vedertagen princip för utveckling av större program är det också lämpligt att testa stora program enligt samma princip.

Normalt användes en kombination av *top-down* och *bottom-up* testning.

Det är vedertagen praxis att utföra enhetstester och systemtest. Man bör dock inte gå direkt från enhetstestning till systemtestning, utan istället successivt utöka systemet med nya programdelar och utföra testningar allteftersom programdelarna integreras i systemet. Detta kallas för *inkrementell testning*.

Paus

15 min

Undantag – exceptionella händelser

Ett undantag (*exception*) är en händelse under exekveringen som signalerar att ett fel har uppstått.

Undantag är vanligtvis svåra att gardera sig emot och ligger ofta utanför programmets direkta ansvarsområde.

Om undantaget inte tas om hand avbryts programmet och ett felmeddelande skrivs ut.

Java har inbyggda mekanismer för att handha undantag.

Vi skall här titta på de mest grundläggande språkkonstruktionerna för undantagshantering.

Orsaker till undantag

Några orsaker till undantag:

- Programmeringsfel (refererar till ett objekt som inte finns, adresserar utanför giltiga index i ett fält, ...).
- Användaren av koden har inte läst dokumentationen (anropar metoder på fel sätt).
- Resursfel (nätverket gick ner, hårddisken är full, minnet är slut, databasen har kraschat, DVD:n var inte insatt, ...).

Hantera undantag

För att handha undantag tillhandahåller Java:

- konstruktioner för att "kasta" undantag (konstruktionen **throw**)
- konstruktioner för att "specificera" att en metod kastar eller vidarebefordrar undantag (konstruktionen **throws**)
- konstruktioner för att fånga undantag (konstruktionerna **try**, **catch** och **finally**).

Felhanteringen blir därmed en explicit del av programmet, d.v.s. felhanteringen blir synlig för programmeraren och kontrolleras av kompilatorn.

Hantering av undantag

När det gäller hantering av undantag kan man ha olika ambitionsnivåer:

- Ta hand om händelsen och försöka vidta någon lämplig åtgärd i programmenheten där felet inträffar så att exekveringen kan fortsätta.
- Fånga upp händelsen, identifiera den och skicka den vidare till anropande programmenhet.
- Ignorera händelsen, vilket innebär att programmet avbryts om händelsen inträffar.

Undantag – exempel

Ett programexempel:

```
import javax.swing.*;
public class Squar {
    public static void main (String[] arg) {
        String indata = JOptionPane.showInputDialog("Ange ett heltal:");
        int tal = Integer.parseInt(indata);
        int res = tal * tal;
        JOptionPane.showMessageDialog(null, "Kvadraten av talet är " + res);
    } //main
} //Squar
```

Vad händer om användaren t.ex. anger "12.3" eller "ett" som indata?

Programmet avbryts och en felutskrift erhålls:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "12.3" at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:458)
at java.lang.Integer.parseInt(Integer.java:499)
at Squar.main(Squar.java:5)
```

Undantag – olika typer

Undantag kan inträffa, som i exemplet ovan, om en användare ger felaktig indata eller om programmeraren tänkt fel och förbisett olika situationer som kan inträffa.

Några vanliga undantag och felen som orsakar dessa:

ArrayIndexOutOfBoundsException

Försök att i ett fält indexera ett element som inte finns

NullPointerException

Försök att anropa ett objekt vars referens har värdet **null**

NumberFormatException

Försök att konvertera en stäng till ett tal, där strängen innehåller otillåtna tecken.

IllegalArgumentExecption

Försök att anropa en metod med ett otillåtet argument.

Kasta undantag - Exempel

Låt oss säga att vi vill skriva en klass **Account** för bankkonton. Vi vill bland annat ha en metod **withdraw** som tar ut pengar ur kontot. Men vad ska vi göra om pengarna inte räcker till?

```
public class Account {  
    private int balance;  
    //flera instansvariabler och metoder som inte visas här  
    public void withdraw(int amount) {  
        if (amount < balance) {  
            balance = balance - amount;  
        }  
        else {  
            // Vad ska vi göra här?  
        }  
    }  
} //withdraw  
} //Account
```

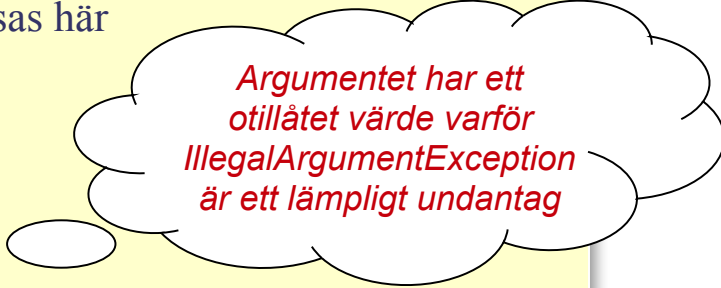
Kasta undantag - Exempel

Ett sätt är att kasta ett undantag. Detta görs med en **throw**-sats enligt följande kodmönster:

```
throw new ExceptionType();  
    där ExceptionType är en existerande undantagsklass.
```

Vårt programexempel blir:

```
public class Account {  
    private int balance;  
    //flera instansvariabler och metoder som inte visas här  
    public void withdraw(int amount) {  
        if (amount < balance) {  
            balance = balance - amount;  
        }  
        else {  
            throw new IllegalArgumentException("Sorry, you are short of money!");  
        }  
    }  
} //withdraw  
} //Account
```



Argumentet har ett otillåtet värde varför IllegalArgumentException är ett lämpligt undantag

Att fånga undantag

För att fånga undantag tillhandahåller Java **try-catch-finally**-satsen.

Kodmönster:

```
try {  
    <Kod som kan kasta (generera) ett undantag>  
} catch (ExceptionType e) {  
    <Kod som vidtar lämpliga åtgärder om undantaget ExceptionType inträffar>  
} finally {  
    <Kod som utförs oberoende av om undantaget inträffar eller inte>  
}
```

Om koden i **try**-blocket inte kastas något undantag av **ExceptionType** ignoreras **catch**-blocket. Om ett undantag av typen **ExceptionType** inträffar avbryts exekveringen i **try**-blocket och fortsätter i **catch**-blocket.

finally-blocket exekveras oberoende av om ett undantag har inträffat eller inte.

finally-blocket kan utelämnas och vi diskuterar inte detta ytterligare på kursen.

Att fånga undantag - exempel

I vårt tidigare programexempel är det möjligt att åtgärda det uppkomna undantaget genom att låta användaren göra en ny inmatning:

```
import javax.swing.*;
public class FaultTolerantSquare {
    public static void main (String[] arg) {
        boolean done = false;
        while (!done) {
            String indata = JOptionPane.showInputDialog("Ange ett heltal:");
            try {
                int tal = Integer.parseInt(indata);
                int res = tal * tal;
                JOptionPane.showMessageDialog(null, "Kvadraten av talet är " + res);
                done = true;
            } catch (NumberFormatException e) {
                JOptionPane.showMessageDialog(null, "Otillåten indata. Försök igen!");
            }
        }
    }
} //main
} //FaultTolerantSquare
```

Att skapa egna undantagstyper

Det finns fördefinierade typer av undantag:

- `ArrayIndexOutOfBoundsException`
- `NullPointerException`
- `NumberFormatException`
- `IllegalArgumentException`
- ...

Man kan också skapa egna typer av undantag genom att skapa subklasser till klassen `RuntimeException` (eller till klassen `Exception`).

```
public class MyException extends RuntimeException {  
    public MyException() {  
        super();  
    }  
    public MyException(String str) {  
        super(str);  
    }  
} //MyException
```

Att skapa egna undantagstyper

```
public class MyException extends RuntimeException {  
    public MyException() {  
        super();  
    }  
    public MyException(String str) {  
        super(str);  
    }  
} //MyException
```

Konstruktorn `MyException(String str)` används för att beskriva det uppkomna felet. Beskrivningen kan sedan läsas när felet fångas m.h.a. metoden `getMessage()`, som ärvs från superklassen. Om felet inte fångas i programmet kommer den inlagda texten att skrivas ut när programmet avbryts.

Även metoden `printStackTrace()`. Denna metod skriver ut var felet inträffade och "spåret" av metoder som sänt felet vidare. Metoden `printStackTrace()` anropas automatiskt när en exceptionell händelse avbryter ett program.

Checked and Unchecked Exceptions

I Java skiljer man på två typer av undantag:

- *unchecked exceptions*, som är subklasser till RuntimeException
- *checked exceptions*, som är subklasser till Exception

De undantag vi sett hittills har varit unchecked exceptions.

Unchecked Exceptions

Unchecked exceptions kan förekomma i princip var som helst och beror ofta på programmeringsfel. Vanligtvis låter man där bli att fånga denna typ av undantag, eftersom det är koden som skall rättas till. Undantaget fångas endast om det orsakas av interaktionen med en yttre användare.

Exempel:

ArithmeticException När resultatet av en aritmetisk operation inte är väldefinierad, t.ex. division med noll.

NullPointerException När man försöker använda en referensvariabel som inte har blivit tilldelad ett objekt

```
Scanner sc;  
sc.nextInt();
```

IllegalArgumentException Används när en metod får ett argument som inte kan behandlas.

Checked Exceptions

Checked exceptions används för att signalera fel som ofta *inte är programmeringsfel*. Dessa fel beror ofta på saker som är utanför programmerarens kontroll. Därför är det viktigt att fånga denna typ av undantag och återställa programmet ett giltig tillstånd eller avbryta programmet på ett kontrollerat sätt.

Exempel:

`FileNotFoundException` när man försöker läsa en fil som inte existerar.

`LineUnavailableException` en ljudenhet man försöker använda är upptagen av ett annat program.

Checked Exceptions

Om man anropar en metod som kan kasta ett checked exception måste man *antingen fånga undantaget eller deklarera att man kastar det vidare.*

Om man vill kasta ett exception vidare i en metod deklarerar man detta genom att lägga till **throws** och namnet på typen för det undantag man vill kasta vidare. Detta skrivs efter parametrarna till en metod.

```
public static void main(String[] args) throws FileNotFoundException {  
    ....  
}
```

När man anropar en metod som kan kasta ett checked exception måste man alltså *aktivt ta ställning till om undantaget skall hanteras eller kastas vidare.*

Kasta checked exceptions vidare

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class ReadFromTextFile {
    public static void main(String[] args) throws FileNotFoundException {
        System.out.println("The sum is: " + sumValuesInFile("indata.txt"));
    } //main

    private static int sumValuesInFile(String fileName) throws FileNotFoundException {
        File in = new File(fileName);
        Scanner sc = new Scanner(in);
        int sum = 0;
        while (sc.hasNext()) {
            sum = sum + sc.nextInt();
        }
        return sum;
    } //sumValuesInFile
} //ReadFromTextFile
```

Kasta exception vidare

*Kan resultera i
FileNotFoundException*

Fånga checked exceptions

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class ReadFromTextFile2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        boolean done = false;
        while (!done) {
            System.out.print("Give filename: ");
            String fileName = sc.next();
            try {
                System.out.println("The sum is: " + sumValuesInFile(fileName));
                done = true;
            } catch (FileNotFoundException e) {
                System.out.println("File doesn't exist!");
            }
        }
    } //main

    private static int sumValuesInFile(String fileName) throws FileNotFoundException {
        //som tidigare
    } //sumValuesInFile
} //ReadFromTextFile2
```

Fånga exception
och vidtag
lämpliga åtgärder

*FileNotFoundException
kan inträffa*

Fler olika typer av undantag kan kastas

```
private static int sumValuesInFile(String fileName) throws FileNotFoundException {  
    File in = new File(fileName);  
    Scanner sc = new Scanner(in);  
    int sum = 0;  
    while (sc.hasNext()) {  
        sum = sum + sc.nextInt();  
    }  
    return sum;  
} //sumValuesInFile
```

Kan resultera i
FileNotFoundException

Kan resultera i
InputMismatchException

`InputMismatchException` är ett av typen *unchecked exception* och finns deklarerad i paketet `java.util`.

Fånga olika typer av undantag

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.InputMismatchException;
import java.util.Scanner;
public class ReadFromTextFile3 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        boolean done = false;
        while (!done) {
            System.out.print("Give filename: ");
            String fileName = sc.next();
            try {
                System.out.println("The sum is: " + sumValuesInFile(fileName));
                done = true;
            } catch (FileNotFoundException e) {
                System.out.println("File don't exist!");
            } catch (InputMismatchException e) {
                System.out.println("Wrong data in file!");
            }
        }
    }
} //main
...
} //ReadFromTextFile3
```



Fångar
FileNotFoundException



Fångar
InputMismatchException

Uppgift

Skriv en metod

```
public static int[] reverse(int[] arr)
```

som tar ett heltalsfält **arr** och arrangerar om elementen i fältet på så sätt att de kommer i omvänd ordning. Om parametern **arr** är **null** skall en **IllegalArgumentException** kastas.

Exempel:

Antag att följande deklARATION har gjorts:

```
int[] f1 = {1, 2, 3, 4};
```

```
int[] f2 = null;
```

anropet **reverse(f1)** skall då returnera fältet [4, 3, 2, 1]

anropet **reverse(f2)** skall då ge ett **IllegalArgumentException**

Lösning

```
import java.util.Arrays;
public class Uppgift {
    public static void main(String[] args) {
        int[] f1 = {1, 2, 3, 4};
        int[] f2 = null;
        System.out.println(Arrays.toString(reverse(f1)));
        System.out.println(Arrays.toString(reverse(f2)));
    } //main

    public static int[] reverse(int[] arr) {
        if (arr == null)
            throw new IllegalArgumentException();
        int[] res = new int[arr.length];
        for (int i = 0; i < arr.length; i = i + 1) {
            res[i] = arr[arr.length - 1 - i];
        }
        return res;
    } //reverse
} //Uppgift
```