Repetition av
OOP- och Javabegrepp

ArrayList<E>

- En lista i vilken man kan lagra objekt
- Implementerar List-interfacet
- Skiljer sig från ett vanligt endimensionellt fält:
  - Dynamisk ⇒ expanderar när den blir full
  - Inga ”tomma” platser, tar vi bort ett element så fyller de andra upp
  - Annorlunda syntax
  - Enklare användning (färdiga metoder för vanliga operationer)
  - Kan inte lagra primitiva typer

ArrayList

Ett fält är en statisk datastruktur, vilket innebär att storleken måste anges när fältet skapas. Fält är alltså inte anpassade för att lagra dynamiska datasamlingar – som under sin livstid kan variera i storlek.

För att lagra dynamiska datasamlingar i ett fält måste man själv utveckla programkod för att t.ex.:

- ta bort ett element ur fältet
- lägga in ett nytt element på en given position i fältet
- öka storleken på fältet om ett nytt element inte ryms.

Klassen ArrayList är en standardklass (av flera) för all lagra samlingar av objekt. ArrayList lämpar sig bättre för dynamiska datasamlingar än endimensionella fält.
ArrayList finns i paketet java.util.

Klassen ArrayList<E>

Metod	Beskrivning
ArrayList<E>()	skapar en tom ArrayList för element av typen E.
void add(E elem)	lägger in elem sist i listan (d.v.s. efter de element som redan finns i listan).
void add(**int** pos, E elem)	lägger in elem på plats pos. Efterföljande element flyttas ett position framåt i listan.
E get(**int** pos)	returnerar elementet på plats pos.
E set(**int** pos, E elem)	ersätter elementet på plats pos med elem, returnerar elementet som fanns på platsen pos.
E remove(**int** pos)	tar bort elementet på plats pos, returnerar det borttagna elementet. Efterföljande element i listan flyttas en position bakåt i listan.

Klassen ArrayList<E>

Metod	Beskrivning
int size()	returnerar antalet element i listan
boolean isEmpty()	returnerar **true** om listan är tom, annars returneras **false**
int indexOf(E elem)	returnerar index för elementet elem om detta finns i listan, annars returneras -1
boolean cointains(Object elem)	returnerar **true** om elem finns i listan, annars returneras **false**
void clear()	tar bort alla elementen i listan
String toString()	returnerar en textrepresentation på formen $[e_1, e_2, \ldots, e_n]$

Anm: Metoderna indexOf och contains förutsätter att objekten i listan kan jämföras, d.v.s. klassen som objekten tillhör måste definiera metoden

public boolean equals(Object obj)

Alla standardklasser, såsom String, Integer och Double, definierar metoden equals.

ArrayList

Klassen ArrayList är en *generisk klass*. För att skapa en lista av klassen ArrayList måste man ange en typparameter som specificerar vilken typ av objekt som skall lagras lagras i listan.

Exempel:

```
ArrayList<String> words = new ArrayList<String>();
ArrayList<Integer> values = new ArrayList<Integer>();
ArrayList<BigInteger> bigValues = new ArrayList<BigInteger>();
ArrayList<Person> members = new ArrayList<Person>();
```

I en ArrayList kan man *endast spara objekt*, d.v.s. en ArrayList kan inte innehålla de primitiva datatyperna (t.ex. **int**, **double**, **boolean** och **char**).

Vill man lagra värden av primitiva datatyper i en ArrayList måste dessa kapslas in i objekt av motsvarande omslagsklasser (*wrapper class*).

Autoboxing och auto-unboxing

Typomvandling sker automatiskt mellan primitiva datatyper och motsvarande omslagsklasser. Detta kallas för *autoboxing* respektive *auto-unboxing*.

Istället för att skriva

```
Integer talObjekt = new Integer(10);
. . .
int tal = talObjekt.toValue();
```

kan man skriva

```
Integer talObjekt = 10;        // autoboxing
. . .
int tal = talObjekt;           // auto-unboxing
```


Förenklad for-sats (for-each loop)

När man vill behandla alla objekt i en samling (t.ex. en ArrayList eller ett en-dimensionellt fält) lämpar sig den förenklade for-satsen:

Genomlöpning av hela samlingarna med den vanliga for-satsen

```
double[] values = new double[100];
ArrayList<String> listan = new ArrayList<String>();
. . .
for (int index = 0; index < values.length; index = index +1)
    System.out.println(values[index]);
for (int pos = 0; pos < listan.size(); pos = pos +1)
    System.out.println(listan.get(pos));
```

Genomlöpning av hela samlingarna med den förenklade for-satsen

```
double[] values = new double[100];
ArrayList<String> listan = new ArrayList<String>();
. . .
for (double v : values)
    System.out.println(v);
for (String str :  listan)
    System.out.println(str);
```

ArrayList

En lista kan också löpas igenom med hjälp av en iterator:

Genomlöpning av hela samlingarna med iterator:

```
ArrayList<String> listan = new ArrayList<String>();
...
Iterator<String> it = listan.iterator();
while ( it.hasNext() ){
    System.out.println(it.next());
}
```


Klassen PhoneBook implementerad med fält

```
public class PhoneBook {
    private Entry[] book;
    private int count;
    public PhoneBook(int size) {
        book = new Entry[size];
        count = 0;
    }
    public void put(String name, String nr) {
        book[count] = new Entry(name, nr);
        count = count + 1;
    }
    public String get(String name) {
        for (int i = 0; i < count; i = i +1)
            if (name.equals(book[i].getName()))
                return book[i].getNumber();
        return null;
    }
}
```

Maximal storlek måste anges

Antalet element måste bokföras

```
public class Entry {
    private String name;
    private String number;
    public Entry(String name, String number) {
        this.name = name;
        this.number = number;
    }
    public String getName() {
        return name;
    }
    public String getNumber() {
        return number;
    }
}
```


Klassen PhoneBook implementation med ArrayList

```
import java.util.ArrayList;
public class PhoneBook {
    private ArrayList<Entry> book = new ArrayList<Entry>();
    public PhoneBook() {
        book = new ArrayList<Entry>();
    }
    public void put(String name, String nr) {
        book.add(new Entry(name, nr));
    }
    public String get(String name) {
        for (Entry e : book)
            if (name.equals(e.getName()))
                return e.getNumber();
        return null;
    }
}
```

(samma)

```
public class Entry {
    private String name;
    private String number;
    public Entry(String name, String number) {
        this.name = name;
        this.number = number;
    }
    public String getName() {
        return name;
    }
    public String getNumber() {
        return number;
    }
}
```


Uppräkningstyper - enum

I bland vill man kunna använda variabler som endast skall kunna anta vissa givna värden:

gender: MALE, FEMALE
state: READY, RUNNING, BLOCKED, DEAD
suit: HEARTS, SPADES, DIAMONDS, CLUBS
season: WINTER, SPRING, SUMMER, FALL
bearing: NORTH, WEST, SOUTH, EAST

I Java kan detta göras genom att deklarera särskilda *uppräkningsklasser*:

```
public enum Suit {
    HEARTS, SPADES, DIAMONDS, CLUBS;
}
```

enum inte class

Räkna upp alla värden (instanser) typen skall ha

Uppräkningstyper - enum

Varje deklarerat (uppräknat) värde i en uppräkningstyp är en instans av klassen.
Varje värde är implicit **public**, **static** och **final**, alltså en *klasskonstant*.

Metod	Beskrivning
int compareTo(E o)	returnerar ett negativt heltal om aktuellt objekt är mindre än argumentet o, 0 om aktuellt objekt och argumentet o är lika och ett positivt heltal om aktuellt objekt är större än argumentet o. Jämförelsen görs enligt den ordning objekten har deklarerats.
boolean equals(E o)	returnerar **true** om o är lika med aktuellt objekt, annars returneras **false**
String name()	returnerar namnet på aktuellt objekt (enligt deklarationen)
int ordinal()	returnerar ordningstalet för aktuellt objekt (enligt deklarationen)
String toString)	returnerar namnet på aktuellt objekt (enligt deklarationen)
static E valueOf(String str)	returnerar objektet med det angivna namnet
static E[] values()	returnerar ett fält innehållande objekten i klassen

Lägga till tillstånd på enum-konstanter

Eftersom **enum** definierar en klass kan man ge tillstånd och beteenden till objekten som tillhör en uppräkningstyp,
Tillståndet hos en instans beskrivs med hjälp av instansvariabler.
Tillståndet för en instans ges som argument till instansen.
En konstruktor för att initiera tillståndet behöver definieras. Konstruktorn får/kan inte vara **public**.

```
public enum Customer {
    CHILD(50),  ADULT(100), SENIOR(60);
    private int price;
    private Customer(int price) {
        this.price = price;
    }
    public int getPrice() {
        return price;
    }
}
```

Argument som anger tillståndet

Konstruktor som sätter tillståndet

Lägga till beteenden på enum-konstanter

```
public enum DayOfWeek {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY;
    public int nrInWeek() {
        return this.ordinal() + 1;
    }
    public DayOfWeek tomorrow() {
        DayOfWeek[] days = DayOfWeek.values();
        return days[(this.ordinal() + 1) % days.length];
    }
     public DayOfWeek yesterday() {
        DayOfWeek[] days = DayOfWeek.values();
        return days[(this.ordinal() - 1) % days.length];
    }
    public static DayOfWeek getDayWithNr(int dayNr) {
        DayOfWeek[] days = DayOfWeek.values();
        return days[dayNr - 1];
    }
}
```

Returnerar vilket ordningsnummer i veckan Den aktuell dagen har

Returnerar dagen efter aktuell dag

Returnerar dagen före aktuell dag

*Returnerar dagen med ordningsnummer **dayNr** i veckan*

enum och switch-satsen:

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

```
public static void tellItLikeItIs(DayOfWeek day) {
    switch (day) {
        case MONDAY:
            System.out.println("Mondays are bad.");
            break;
        case FRIDAY:
            System.out.println("Fridays are better.");
            break;
        case SATURDAY:
        case SUNDAY:
            System.out.println("Weekends are best.");
            break;
        default:
            System.out.println("Midweek days are so-so.");
    }
}
```

Utförs för SATURDAY och SUNDAY

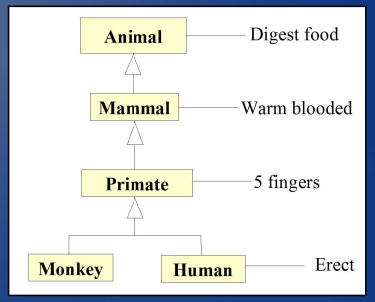
Utförs för TUESDAY, WEDNESDAY och THURSDAY

Implementationsarv

- I Java kan en klass vara en utökning (extends) av en annan klass.
- Alla klasser (utom Object) är i grunden utökningar av klassen Object.
- När en klass utökar en annan så ärver den alla dess fält och metoder.
- En subklass utökar superklassen
- Klasser är del i en klasshierarki

java.lang
Class Integer

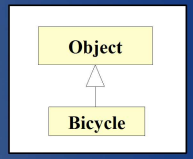
java.lang.Object
└ java.lang.Number
└ java.lang.Integer



Implementationsarv

Vi börjar med att definiera klassen Bicycle, som representerar cyklar:

```
public class Bicycle {
    private int speed;
    public Bicycle(int speed){
        this.speed = speed;
    }
    public void setSpeed(int newSpeed){
        speed = newSpeed;
    }
}
```

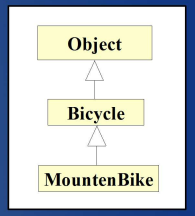


Enda egenskap för ett objekt av Bicycle är att den har en viss hastighet.

Implementationsarv

Vi definierar nu en mer avancerad cykel (MountainBike) som ärver egenskaperna från Bicycle, och lägger till ytterligare funktionalitet, en MountainBike har växlar:

```
public class MountainBike extends Bicycle {
    private int gear;
    public MountainBike(int startSpeed, int startGear){
        super(startSpeed);
        gear = startGear;
    }
    public void setGear(int newGear){
        gear = newGear;
    }
}
```



Vi behöver alltså inte implementera de egenskaper som Bicycle har, utan implementationen (koden) från klassen Bicycle ärvs.

Implementationsarv

- MountainBike är en (direkt) *subklass* till Bicycle.
- Bicycle är (direkt) *superklass* till MountainBike.
- Bicycle och MountainBike är *subklasser* till Object.
- Object är superklass till Bicycle och MountainBike.
- Varje klass introducerar en ny typ, med samma namn som klassen.
- Object, Bicycle och MountainBike är typer.
- Typen MountainBike är en subtyp till typen Bicycle.
- Typen Bicycle är en supertyp till typen MountainBike.
- Varje MountainBike *är* en Bicycle
- En Bicycle är INTE en MountainBike

Implementationsarv

I Java gäller att:
där det efterfrågas ett objekt av datatypen C får man även ge ett objekt av en subtyp till C

- Om någon, ex. en metod, kräver en Bicycle är det ok att ge en MountainBike, men det omvända gäller ej.

```
public static void park(Bicycle bike){
   // code not shown
}

MountainBike myBike = new MountainBike(40, 12);
park(myBike);
```


Implementationsarv: super

En subklass kan referera till sin direkta superklass med **super**

- används i subklassens konstruktorer för att anropa superklassens konstruktorer

  super(startSpeed);

  Om subklassens konstuktor gör anrop till superklassens konstruktor skall detta anrop ligga som första sats i konstruktorn.

  Gör inte subklassens konstruktor något anrop till superklassens konstruktor sker automatiskt anrop till superklassens default-konstruktor (konstruktorn utan parametrar):

  - saknar superklassen helt konstruktorer propagerar anropet uppåt i klasshiearkin
  - saknar superklassen default-konstruktor men har någon annan konstruktor uppkommer ett kompileringsfel.

Implementationsarv: super

- används i subklassen för att anropa överskuggade metoder i superklassen

```
public class Bicycle {
   // som tidigare
   @Override
   public String toString() {
      return "Speed: " + speed;
   }
}
```

```
public class MountainBike extends Bicycle{
    public int gear;
   // som tidigare
   @Override
   public String toString() {
      return super.toString() + ", Gear: " + gear;
   }
}
```

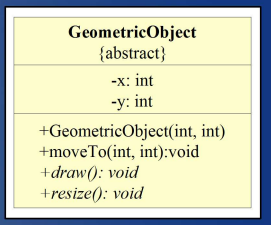
En överskuggad (*override*) metod är en *instansmetod* som omdefinieras i subklassen.

Implementationsarv: Abstrakta klasser och metoder

En abstrakt klass en en klass som innehåller en eller flera abstrakta metoder.

En abstrakt metod är en metod som saknar implementation.

```
abstract public class GeometricObject {
    private int x, y;
    public GeometricObject(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void moveTo(int newX, int newY) {
        x = newX;
        y = newY;
    }
    abstract public void draw();
    abstract public void resize();
}
```



Dessa metoder ser olika ut beroende på figuren, så vi kan inte här säga exakt hur implementationen ser ut. Vi låter dem förbli abstrakta.

Implementationsarv: Abstrakta klasser och metoder

- Det går inte att skapa instanser av en abstrakt klass.
- Abstrakta klasser används för att abstrahera en samling klasser som har många gemensamma egenskaper, men som även har vissa olikheter.
- Abstrakta klasser implementerar vissa metoder som är gemensamma för alla subklasser.
- De abstrakta metoderna måste implementeras av subklasserna (såvida inte även dessa är abstrakta).

Implementationsarv: Abstrakta klasser och metoder

Vi kan ha en ”samlingsklass” (ex. GeometricObject), som ärvs av flera subklasser (ex. Rectangle, Circle).

Om vi låter GeometricObject vara en vanlig klass, som vi gjorde med Bicycle, så går det att skapa instanser av den. Det vill vi dock inte tgillåta, då GeometricObject inte är en faktisk geometrisk form.

Lösningen kan då vara att göra den till en abstrakt klass. Man kan då inte skapa instanser av den, men den kan fortfarande innehålla vissa gemensamma egenskaper och den kan fortfarande ärvas.

Implementationsarv: Abstrakta klasser och metoder

GeometricObject är inte en faktisk geometrisk form, utan definierar gemensamma egenskaper som geometriska former har.

Verkliga geometriska former utökar klassen GeometricObject genom att överskugga de metoder som är abstrakta:

```
public class Rectangle extends GeometricObject {
    @Override
    public void draw() {
        // kod
    }
    @Override
    public void resize() {
        // kod
    }
}
```

```
public class Circle extends GeometricObject {
    @Override
    public void draw() {
        // kod
    }
    @Override
    public void resize() {
        // kod
    }
}
```

Här får subklassen specificera hur just den väljer att implementera de abstrakta metoderna.

För de metoder som är implementerade i GeometricObject ärver subklasserna koden.

Specifikationsarv: Interface (gränssnitt)

- I Java kan en klass endast ärva från (göra **extends** på) en klass.
- I Java åstadkoms multipelt arv via specifikationsarv m.h.a. av interface.
- Ett interface saknar implementation, d.v.s. alla metoder som specificeras är abstrakta.
- En metod som ärver från ett interface implementerar metoderna som interfacet specificerar.
- En klass kan implementera flera interface.
- Varje interface introducerar en ny typ, med samma namn som interfacet.
- Varje klass som implementerar ett interface är en subtyp till typen för interfacet.
- Klasser och interface bildar en typhierarki.

Specifikationsarv: Interface

Exemplet ActionListener:

För att en klass ska kunna lyssna och reagera på händelser måste den vara redo att ta emot händelser. Men hur ska andra klasser kunna lita på att denna klass har implementerat de nödvändiga metoderna för att lyssna på händelser? Vad heter dessa metoder? Vilka indata skall metoderna ha?

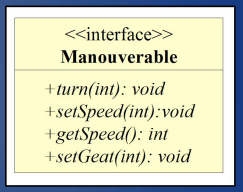
Lösningen är att använda interfacet ActionListener. Alla klasser som implementerar ActionListener är tvungna att implementera nödvändiga metoder, som ActionListener har specificerat, för att ta emot händelser. Klassen blir därmed en ”certifierad” ActionListener och andra klasser kan med gott samvete lägga till den som lyssnare. I detta fall agerar ett interface alltså som ett avtal mellan olika klasser.

Specifikationsarv: Interface

Syntax för gränssnitt:

Alla klasser som vill tillhöra kategorin motorfordon måste implementera interfacet Manouverable, för att säkerställa att de har grundläggande funktionalitet för manövrering.

```
public interface Manouverable {
    void turn(int degrees);
    void setSpeed(int newSpeed);
    int getSpeed();
    void setGear(int newGear);
}
```

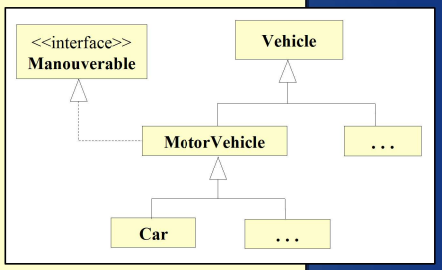


Specifikationsarv: Interface

Syntax för implementation:

En klass som implementerar gränssnittet måste implementera alla dess metoder.

```
public class MotorVehicle extends Vehicle implements Manouverable {
    . . .
    public void turn(int degrees){
        // kod
    }
    public void setSpeed(int newSpeed) {
        // kod
    }
    public int getSpeed() {
        return speed;
    }
    public void setGear(int newGear) {
        // kod
    }
}
```



Metoden equals()

Klassen Object specificerar bl.a. följande metod:

boolean equals(Object o) — jämför om objektet är lika med ett annat objekt.

Alla klasser bör/skall överskugga denna metod.

```
public class Bicycle {
    private int speed;
    //som tidigare
    @Override
    public boolean equals(Object other) {
      if (this == other)
          return true;
      if (other == null)
          return false;
      if (other.getClass() != this.getClass())
          return false;
      Bicycle otherObject = (Bicycle) other;
      return speed == otherObject.speed ;
    }
}
```

```
public class MountainBike extends Bicycle{
    public int gear;
    //som tidigare
    @Override
    public boolean equals(Object other) {
        return super.equals(other) &&
               gear == other.gear;
    }
}
```

Vad är exceptions?

- Ett exception (undantag) är ett objekt som representerar ett fel.
- Undantatg kan deklareras, skapas, kastas, fångas och hanteras.
- Många standardmetoder skapar och kastar undantag om något fel inträffar.
- Det finns olika undantagsklasser, exempelvis:
  - NullPointerException
  - NumberFormatException
  - ArrayIndexOutOfBoundsException
- API:n beskriver vilka undantag en metod kan kasta.

try/catch

Där vi anropar en metod som kan orsaka ett undantag, kan vi införa try/catch-block för att fånga upp dem:

```
try{                                    // Försök att göra det här.
    // kod...
    int i = Integer.parseInt(str);      // Kan orsaka ett NumberFormatException
    // kod...
} catch(NumberFormatException e) {// Om ett NumberFormatException dyker
                                        // upp, fånga det och döp det till "e".
    e.printStackTrace();                // Skriv ut ett spårbart felmeddelande

    // Gör eventuellt fler åtgärder.
}
```


Vad används undantag till?

- Hantera fel som kan uppstå under körning (runtime).
  - Användaren slipper se en krasch.
  - Vi kan försöka lösa problemet.
  - I värsta fall: användaren får ett fint avsked.
- Undvika att programmet kraschar.
- Kan delegera ansvaret för att hantera felet
  – felet måste inte hanteras där det uppstod.

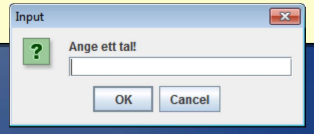
Vad är ett fel?

- En situation som normalt inte ska inträffa och som är svår att undvika i förväg, exempelvis:
  - Öppna en fil som inte existerar
  - Använda ett index i ett fält utanför dess gränser
  - Division med noll
  - Inmatningsfel, ex. en bokstav istället för en siffra

Exempel på ett fel

Kalle har skrivit en metod som frågar efter ett tal:

```
private static int askNumber(){
    String input = JOptionPane.showInputDialog("Ange ett tal!");
    int number = Integer.parseInt(input);
    return number;
}
```



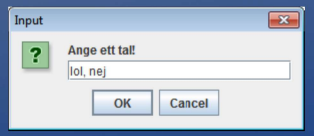
Vad kan gå fel?

Exempel på ett fel

När vi skriver koden vet vi inte vad användaren kommer att ge för indata!

```
private static int askNumber(){
    String input = JOptionPane.showInputDialog("Ange ett tal!");
    int number = Integer.parseInt(input);
    return number;
}
```

Vad händer om input inte är ett tal? Programmet kraschar.



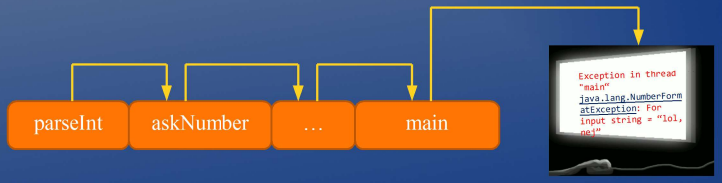

```
Exception in thread "main"
java.lang.NumberFormatException:
For input string = "lol, nej"
```


Exceptions

Vad gick fel?

Metoden parseInt(input) kunde inte konvertera strängen till ett tal, varför metoden skapade och skickade iväg en NumberFormatException.

Undantaget fångades inte på vägen, varför det fortsatte uppåt i kedjan av anrop, tills den kom till användaren ⇒ krasch



Exceptions

Vad göra?

- Vad vill vi ska hända om användaren ger fel input?
  - Vi varnar användaren och frågar igen!
- Var ska undantaget fångas upp?
  - I det här fallet kan det fångas i askNumber(). Det är denna metod som sköter inläsningen från användaren.
  - Obs! Man fångar inte alltid upp felet där det uppstod. Felet skall fångas upp där vi faktiskt kan göra något åt det.
- Hur fångar vi upp undantag rent praktiskt?
  - Ett try/catch-block som omger den riskfyllda koden.

Exceptions

Hantering av undantag.

```
private static int askNumber() {
    int number;
    while (true) {
        try {
            String input = JOptionPane.showInputDialog("Ange ett tal!");
            number = Integer.parseInt(input);
            break;
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Felaktigt tal!");
        }
    }
    return number;
}
```

När användaren trycker på cancel-tangenten ges en felutskrift och en ny inläsning begärs. Är detta korrekt?

Exceptions

Vidarebefordran av undantag.

```
private static int askNumber() throws NumberFormatException {
    String input = JOptionPane.showInputDialog("Ange ett tal!");
    return Integer.parseInt(input);
}
```


Exceptions

- Undantag är ett smidigt sätt att hantera och kontrollera fel.
- Vi kan använda undantag till att göra programmet mer funktionellt och användarvänligt.
- Många färdiga metoder i Java kan kasta undantag.
- Vi kan själva fånga, hantera eller kasta vidare olika undantag.
- Vi kan också skapa undantag och kasta dem.
- Vi kan definiera egna undantagsklasser.