# Java-grunder

OPA 20

# Inledning

Denna text är en sammanfattning av grundläggande Java som vi gjort hittills i OPA-20. Texten är främst gjort för egen referens, men kan användas av dom som vill. Detta är på inget sätt en officiell dokumentation av Java, mer en sammanfattning av egna tankar kring Java och dess funktioner.

Detta dokument är för tillfället inte helt klart och befinner sig i version 0.8. 1.0 släpps förhoppningsvis snart, med input från kurskamraterna.

# Java

Java är ett språk med en lång historia. Java utvecklades först av företaget SUN på 1990-talet, men sedan SUN köptes upp 2010 företaget Oracle - driver nu Oracle utvecklingen av språket. Java är för närvarande det tredje mest använda språket i världen när det kommer till programmering.

Java har två delar, den ena handlar om programmeringsspråket i sig, den andra handlar om maskinkod. Programmeringsspråket Java är utformat för att det ska vara lättare att människor att skriva program. Dock räcker det inte för datorer, som behöver maskin-kod.

Detta görs genom att man skapar sina program i programmeringsspråket och sedan kompilerar man koden till maskinkod via Java-compiler. De filer som tidigare då varit .java blir nu .javac. Koden körs och tolkas av en virtuell maskin som kallas för JVM (Java Virtual Machine), som översätter maskinkoden för datorn. Språket, kompilatorn och JVM finns inom utvecklarverktyget JDK (Java Development Kit).

JDK är open-source, vilket innebär att den är fri att använda som man önskar för eget bruk. Dock är JDK en öppnare version av Oracles Java SE (Standard Edition). Java SE är en licens för kommersiellt bruk. Både Java SE- och Java EE (Enterprise Edition) licenser kostar pengar och det är först när man har dessa licenser som man får skriva kommersiella applikationer.

# Kommentarer

Ibland behöver man kommentera kod eller tillfälligt kommentera ”bort” kod för testning. Detta görs på två sätt.

### Single line comment

En single line comment sker bara på en linje och kommenterar ut den. Detta görs med dubbla //.

Exempel:

*// String denhärsträngenkommerattvarautkommenterad = ”Dethärmed”;*

### Multi-line comment

En multi-line comment sker på flera linjer i koden och är bra om man vill koda ut hela block av kod. Detta görs med en öppning: /* och en stängning */

Exempel:

*/* Allt inom det*

*här blocket*

*kommer att*

*kommenteras ut */*

# Operator

Operatorer är de matematiska och logiska verktygen som finns för att räkna och jämföra data. Dessa operatorer är inbyggda i Java och återfinns i princip alla programmeringsspråk. Följande är de mest grundläggande uttrycken.

De mest vanliga matematiska uttrycken:

| | |
|---|---|
| + | Addition |
| - | Subtraktion |
| * | Multiplikation |
| / | Division |
| % | ”floor division” (rest-division) |
| ++ | öka med 1 |
| -- | minska med 1 |

Matematiska uttryck följer operatorprioritet (PUMDAS). Det finns även mer avancerade matematiska funktioner i Javas inbyggda Math-bibliotek.

Boolska uttryck:

true
false

Jämförelser:

| | |
|---|---|
| == | Exakt lika med |
| != | Inte lika med |
| > | Större än |
| < | Mindre än |
| >= | Större eller lika med än |
| <= | Mindre eller lika med än |

Logiska operatorer

| | |
|---|---|
| && | OCH |
| \|\| | ELLER |
| ! | INTE |

Operatorer kan användas i sammanhang där man vill skapa Boolean-värden, men är även bra när man vill jämföra tal och matematiska uttryck. För att jämföra text/strängar (Strings) behövs speciella verktyg som är inbyggda i String-klasser/typer.

# Variabler

Variabler är små containers av information som sparas i minnet medan programmet körs. En variabel kan vara Primitiv eller icke primitiv. Eftersom Java är ett väldigt strikt språk, så måste alla variabler deklareras, dvs att varje variabel måste vara en typ. Pimitiva typer måste alltid ha ett värde. Icke primitiva typer kan ha ett ”null”-värde.

### Access modifiers

En variabel kan vara publik (public), privat (private) eller skyddad (protected). En variabel som är public kan nås från var som helst, även andra klasser. En privat variabel kan bara användas av klassen den är skriven i. En skyddad variabel kan bara användas av kod som tillhör samma paket (package) och subklasser till variabelns klass.

### Primitiva typer (Primitive types)

Primitiva typer är inbyggda i Java. Dessa typer är reserverade och deras funktioner kan inte omdefinieras.

**Primitiva typer är:**

---

**Integer types**

byte - Heltal (1 byte)

short - Heltal (2 bytes)

int - Heltal (4 bytes)

long - Heltal (8 bytes)

**Floating types**

float - Decimaltal (4 bytes)

double - Decimaltal (8 bytes)

**Övriga**

boolean - true/false (1 bit)

char - Karaktär (2 bytes)

---

## Type-casting

Eftersom Java är så strikt i sitt syntax, så är en typ väldigt strikt hållet till en typ genom hela programmet. Om man behöver använda informationen i sin variabel till något annat och som en annan typ, så måste man göra en Typecast.

Grundregeln är:

- **En mindre typ castas automatiskt till en större.**

  Exempel:

  ```
  int heltal = 9;
  double omvandladVar = heltal;     // heltalet kan nu användas som omvandladVar
  ```

- **En större typ måste manuellt castas till en mindre.**

  Exempel:

  ```
  double ettDecimaltal = 5.4;
  int ettHeltal = (int) ettDecimaltal   // double omvandlas till int och avrundas
                                        automatiskt
  ```

---

## Non-primitive types

Non-primitive types är saker som kan kalla olika metoder och ofta är definierad av användaren. Till dessa hör metoder, klasser, arrays, interfaces mfl. men även String räknas hit.

Non-primitive types kännetäcknas oftast av att de inleds med stor bokstav och går att definiera av användaren.

## ”Scope”

Inom programmering pratar man ofta om Scope för att beskriva en variabels position. Positionen är viktig, då en variabel kan vara lokal, publik och/eller tillfällig. En

# Simpel input (Scanner)

För att ta in data, finns det ett antal metoder för att hantera det. En vanlig metod för detta är att ta in data med en Scanner. En Scanner är en inbyggd metod som tar in input och sedan lämnar ifrån sig den typ av information som specificeras av användaren.

En scanner skapas först genom att man deklarerar den som ett nytt Scanner-objekt. Därefter sparas inmatningen i minne, för att sedan definieras i en kommande variabel. Som i exemplet nedan där ”scanner” sparar en inmatning, som sedan lagras som en integer (genom scanner.nextInt( ) ) i variabeln aNum.

Exempel:

*Scanner scanner = new Scanner;*
*int aNum = scanner.Nextint();*

# Simpel Output (SOUT,SOUF m.fl.)

För att skriva ut information så behövs en metod för utskrifter. Javas inbyggda metoder hanterar System-out-metoder, vilket innebär att den använder operativsystemets egna metod för att skriva ut saker till en ”terminal”.

## SOUT

SOUT är kortkommando för System.out.println(innehåll). Detta gör en ny rad och skriver ut innehållet som matats in.

Exempel:

*int anInt = 3;*

*String aString = ”4”*

*System.out.println(aString+ anInt );*

*Detta är en metod som skriver ut en kombination av en sträng och en variabel.*

*Outputen blir:*

*43*

## SOUF

SOUF är förkortningen för System.out.printf();

Printf gör att man kan formatera en sträng enklare med hjälp av ett % med typen av variabel (%s är en sträng, %d är en digit/nummer m.m.) följt av variablerna i fråga.

*Exempel*

*String enSträng = ”talet”*

*int ettTal = 5;*

*System.out.printf(”Första %s är %d”, enSträng, ettTal)*

*Output:*

*”Första talet är 5”*

**Print**

Det finns en uppsjö av andra outputs att välja från. Dock är de två övre och ”print” bland de mer vanliga.

```
System.out.print();   // skriver ut, men skapar inte en ny rad
```

# Klasser

Klasser är som ”formar”/blueprints till objekt och kan användas för att skapa multipla och oftast identiska objekt. Klasser kan innehålla variabler, attribut och metoder som är specifika för klassen, men kan även anropa attribut från andra klasser om så skulle behövas. En klass kan även ärva attribut och metoder från en annan klass (mer om det i kapitlet Arv, senare i det här dokumentet).

Varje klass är en egen fil. Varje fil är kopplat till sitt paket (package), vilket definieras högst upp i varje klass-fil.

Varje Java-projekt måste ha minst en Main-klass och en main-metod för att kunna köra kod. Dock brukar man lägga all specifik kod för ett objekt i sina egna klasser.

Klassfiler kännetecknas av att de slutar på .java.
En klass-fil innehåller alltid följande kod:

---

*package com.example*

*public class ClassName {  //… här skriver man sin kod… }*

---

Även om man oftast kör huvud-main-metoden i sin Main-klass, så kan man också ha en main-metod i en egen klass, för att bara testa klassen. Detta är dock inte direkt standard. I majoriteten av fallen så kör man sin kod i main-metoden i Main-klassen.

## Metoder

Varje klass har sin/sina metoder. En metod innehåller ”mekanik” som automatiskt löser ett antal definierade ”problem”. Dessa problem kan vara allt från att räkna ut ett enkelt tal till att ta in information från användaren av programmet för att sedan processa det på något sätt.

En metod gör en sak. En metod kan innehålla flera mindre metoder för att lösa ett problem.

- Exempel på en metod som beräknar kvadraten av ett tal X.

---

```
public void squareNumber(){

        int x = 2;
        int theSquare = x*x;
        System.out.println(theSquare);      // skriver ut x * x
}
```

---

För att köra den här koden kan man nu ”kalla” på metoden från vår Main-klass (som alltid måste finnas):

```
squareNumber();
```

---

## Argument

En metod kan ta in ”argument”, det vill säga variabler eller objekt som är viktiga för metoden. Man kan till exempel skicka in ett tal eller ett helt objekt som metoden kan räkna, spara eller jämföra med något annat. Här är möjligheterna i princip oändliga. Det enda man behöver göra är att definiera sitt argument i prentesen vid metoden och sedan kan man använda det.

- Exempel: en metod som tar in ett nummer, som sedan adderas med numret 5.

---

```
public void addByFive(int num){

        int sum = num+5;
        System.out.println(sum);
}
```

## Returer

En metod kan returnera ett värde för senare bruk i en annan metod eller process. **Om inget ska returneras** använder man uttrycket ***void*** vid sin metod. I annat fall måste man definiera den Typ som ska returneras.

- Exempel: en metod som ska returnera en Integer. Returen (int) definieras precis innan metoden.

---

```
public int numberCrusher(int numToCrush){

        int smash = numToCrush*0;                   // krossar numret till noll
        int forLater = numToCrush+1;                // definierar ett nummer för retur
        System.out.println("Totally crushed "+ numToCrush);   // skriver ut till skärm


return forLater;        // måste finnas med, då vi ska returnera forLater, som är en int.

}
```

# Arrays

Om variabler sparar enskilda typer av data, så sparar arrays samlingar av variabler och typer av data. Arrays är en form av lista som man kan använda för att spara olika saker. Just nu pratar vi bara om två typer av listor: Array och Arraylist. Gemensamt för båda typerna är att varje ”plats” i listan kallas för index, vilket alltid börjar på talet 0.

## Array

En array är en striktare form av lista, som måste ha en längd och en typ. En array får bara ha en typ och den får inte vara större än sin längd.
För att definiera sin lista så måste man deklarera den enligt följande.

- En array av String-typ, sparad i variablen books, vars längd är 4. Vi lägger sedan in fyra böcker i listan.

---

```
String[] books = new String[4];
books[0] = ”Metro 2033”;
books[1] = ”Emil I Lönneberga”;
books[2] = ”Great expectations”;
books[3] = ”Leviathan”;
```

---

Notera att första ”platsen” i listan är 0. Det är det första talet. Nu ligger alla böcker i listan. För att nå dom, så behöver man en iterator och/eller en for-loop (som i sig är en iterator).

## Arraylist

En Arraylist är en mer dynamisk lista. Den behöver fortfarande en typ, men dess längd kan variera. För att deklarera den så använder man sig av Javas inbyggda Arraylist-typ.

För att skapa en ny lista och lägga till saker i listan gör man enligt följande:

---

```
Arraylist<String> cars = new Arraylist<>( );


cars.add(”Tesla”);

cars.add(”Saab”);

cars.add(”Volvo”);
```

---

Om man bara vill skriva ut innehållet i listan kan man använda sig av en SOUT och en toString-metod:

*System.out.println(cars.toString());*

*Output:*

*[Tesla, Saab, Volvo]*

---

Annars behöver man även här en for-loop för att nå de enskilda elementen i listan. Arraylist har ett antal användbara metoder som man kan använda för att jämföra och hämta ut viss information om element eller om listan i sig självt. Viktiga metoder är:

| | |
|---|---|
| add( ); | lägger till ett element i listan |
| remove( ); | tar bort ett element |
| sort( ); | sorterar listan |
| contains ( ); | returnerar en boolean om elementet i listan finns. |

osv.

För djupare förståelse rekommenderar jag att läsa mer på W3Schools om Arrays och Arrayslists.

# Logik

För att ”styra” kod till att passa speciella villkor, använder man sig av logiska ”gates” som alla har sina olika användningsområden.

### If-sats

En if-sats används om man vill kolla om ett visst villkor och uttryck är sant eller falskt. Ett villkor består av ett uttryck som genererar en Boolean. Om bara ett villkor testas, så körs koden om den uppfyller det villkor som testas, annars görs inget. Om flera villkor testas så jämförs villkoren enskilt tills dess att villkoret mötts eller tills koden möts av en else-klausul. Om else-klausulen inte existerar och uttrycket går rakt igenom if-satsen, så går uttrycket/villkoret oberört igenom hela if-satsen.

Flera villkor som ska testas gör detta med en else if-klausul, som kollar ett villkor som gått vidare från den första if-satsen. Villkoren testas en och en, i den ordning de följer varandra.

Villkor som testas kan använda samtliga jämförelse-operatorer, matematiska operatorer och logiska operatorer för att generera en boolean.

If-satsen deklareras med ”if”, följt av en parentes där villkoret testas, därefter ett kodblock (måsvingar) där kod körs om villkoret möts. Som default testas koden utifrån om den returnerar boolean-värdet true.

Exempel 1:

```
if (2<5){
        System.out.println(”Very true.”);
        // Körs, då 2 är mindre än 5.
}
```

Exempel 2:

```
int num = 3;
if (num < 1){
        return true;
        // Villkoret är tyvärr inte sant, koden körs inte.
}else if (num >7){
        System.out.println(”Very nice!”);
        // Inte heller här är koden sann.
}else{
        System.out.println(”This is it”);
        // Detta kommer köras.
}
```

### While-loop

En While-loop körs så länge ett villkor uppfylls. Som default är villkoret true.

Exempel:

```
    int num = 10;
    while (num < 10){
        System.out.println("This will print 10 times");
        num++;
    // Medan numret är mindre än 10, kör koden. num++ ökar num hela tiden,
    // så koden kommer köras till dess att villkoret inte längre uppfylls.
}
```

Om man inte uppdaterar ”num” i exemplet ovan, så kör loopen oavbrutet. Var varsam med detta! Se till att alltid ha ett villkor som bryter loopen.

### Do-while-loop

En Do-while-loop är en while-loop som ska köras minst en gång, oavsett om villkoret som testas är sant eller falskt. Samma mekanik som styr en while-loop gäller för en do-while-loop, bara det att den körs en gång - oavsett vad som testas.

```
    int conditionTest = 4;
    do {
        // kod som ska köras
        // trots att koden faktiskt är 4.
    }
    While (conditionTest != 4);
```

### For-loop

En for-loop kan användas på flera olika sätt. Den är mest användbar till listor, men också till uttryck som har en bestämd längd. I vissa användningsområden är For-loopen lik While-loopen, då båda kan användas till att köra en loop tills den uppfyller ett villkor.

For-loopen består av en deklaration med ”for”, följt av en parentes som testar ett uttryck. Parentesen i sig består av en variabel, ofta en tillfällig variabel som enbart tillhör for-loopen, därefter kommer ett uttryck som testar ett uttryck som möter ett boolskt värde. Den sista ”platsen” i parentesen är ofta en ”increment” som ökar värdet på första värdet. Därefter kommer kodblocket som körs medan uttrycket i For-loopen är uppfyllt.

- Exempel:

```
for (int number = 0; number < 7; i++ ) {


        System.out.println(number);
        // kommer att skriva ut variabeln number
        // sju gånger (då 0 är första numret)
        // därefter slutar loopen
}
```

Loopen skapar en tillfällig variabel ”number”, testar sedan om number är mindre än 7 och efter varje loop ökar den number med 1. Loopen körs så länge number är mindre än 7.

- Exempel 2: Listor

[ Vi antar att vi redan gjort en lista ”cars” som har längden 4]

```
for (int i = 0; i< cars.length; i++){
        System.out.println(cars[ i ]);
}
```

Loopen skapar den tillfälliga variabeln i, som testas emot cars-listans längd och index. Variabeln ”i” ökar med 1 för varje loop vilket göra att listans index kan representeras i kodblocket. I kodblocket körs en print, som skriver ut elementet i listan på den plats ”i” som for-loopen för tillfället representerar. När variabeln ”i” har nått talet som är längden på cars-listan, så slutar loopen.

## For each

Ett lite enklare sätt att hantera listor, utan att behöva oroa sig om listans längd, är att använda en for-each-loop. For-each-loopen, likt for-loopen, skapar en tillfällig variabel, som sedan körs mot en lista.

• Exempel:

[Vi antar att vi redan skapat en lista av Integers som innehåller id-nummer]

```
for (int i : idNums){
        System.out.println(i);
        // skriver ut varje element (i det här fallet integers)
        // till dess att listan tar slut.
}
```

## Switch

En switch testar ett uttryck mot ett flertal ”case” fram till dess att switchen tar slut, möter ett ”break” eller möter ett ”default”. Ett break bryter listan så fort uttrycket kommit till det ”case” som har break och som uppfyller uttrycket. default läggs i slutet av listan och körs om uttrycket inte uppfylls.

Exempel:

```
int hour = 12;

switch (hour){

case 10:
        System.out.println(”Clock is 10”);
case 11:
        System.out.println(”Clock is 11”);
case 12:
        System.out.println(”Clock is 12, time for lunch!”);
        break;
}
```

---

*Output:*

*Clock is 12, time for lunch!*

## ”Nested logic”

I vissa fall behöver man kombinera två eller flera olika typer av loopar eller if-satser. Då kan man göra något som kallas för nästade loopar/satser. Alla kombinationer är möjliga, det är lösningen på ”problemen” som avgör hur man nästar logik.

Exempel: En if-sats som leder till en for-loop om en sträng finns i en lista

[vi antar att följande uttryck är sant, att elementet finns i listan ”bikes”]

```
if (bikes.contains(”Kronans”)){
        for (String i : bikes){
                // Kör den här koden
        }
}
```

Exempel 2: En for-loop som innehåller en if-sats. For-loopen går igenom alla element i listan, skickar dom en och en till if-satsen, som kollar om elementet finns i listan. Om elementet finns, skriver den ut ”Found it” och tar bort det elementet från listan. Därefter bryter loopen genom ”break”.

```
for (String i : bikes){
        if (i.contains(”Kronans”)){
                System.out.println(”Found it!”);
                bikes.remove(i);
                break;
        }
}
```

# Mer om klass

Som tidigare nämnts så har varje klass sin specifika attribut och metoder. Till detta kommer ett antal metoder som är av vikt för att skapa och koppla ihop klasser.

### Constructor

En konstruktor är bland det första som körs när man skapar ett objekt.  Det kallas att man initialiserar klassen. Konstruktorn behöver inte ha argument, men om den har argument så kan man använda fältet till att deklarera nödvändiga attribut.

### Getters/Setters

### Arv

### Super

### Composition

### Aggregation

# Try/Catch