

# Introduction to Java

Mikael Svahnberg\*

2024-02-19

## 1 Introduktion

- Snabbintroduktion till *Objektorienterad Programmering*
- Ett exempel på ett kompilerat programspråk: Java
  - Kompileringscykel
  - Klasser, Objekt, Attribut
  - Villkor, Iterationer
  - Paket
- Titta gärna på Youtube-serien “Java for the Reluctant”
  - <https://www.youtube.com/playlist?list=PLRyU9jMTRIX7QUrxF9HNLbdXtiv6F9MT>
  - <https://github.com/mickesv/Reluctant-Java.git>

## 2 Object Oriented Programming

### 2.1 Background: Data Representation and Manipulation

- Computer Programs are essentially about *Data*, e.g.
  - Text
  - Bank Account Information
  - Demographics
  - Warehouse Inventory
  - Medical Journals
  - Collections of Live or Historical Measurements from some device
  - Images, Video, ...
  - Maps
  - ...
- Some data only exist within the realm of a computer program

---

\*Mikael.Svahnberg@bth.se

- Other (most) data represent something in the real world

Challenges

- *Represent* the data accurately
- *Manipulate* the data consistently
- *Store* data sufficiently durably

## 2.2 Representing Data

- Simple datatypes: *int*, *float*, *char*
- Compound datatype: *person*, *bank account*, *text document*, *position*

Challenge

- We usually do not only store *one* datum; we have collections of data, e.g.
  - *temperature readings every minute for the past 24 hours*
  - *persons currently employed by this company*
  - *patients in Sweden*

## 2.3 Manipulate Data

- Program Structuring ensure that all code that manipulates a certain type of data is found in the same place.
  - All code that manipulates *X* is in this directory.
  - All code that manipulates *X* is in this file.
  - All code that manipulates *X* have the moniker **X** in their name.
  - ...
  - *Only* code with an **X** in their name is allowed to operate on *X*.
  - code with an **X** in their name is only allowed to operate on a single instance of *X*.
  - ...
- The data representation may make it easier or harder to manipulate the data *consistently*.
  - All parts of *X* start with **X** , e.g. **PersonName**, **PersonPhoneNumber**, **PersonAddress**,
  - Position **i** in all collections represent part of the same datum, e.g. **PersonName[1]** has the corresponding **PersonPhoneNumber[1]**.

Or, we may take a different approach.

## 2.4 Objects: Representing Compound Data Types

```
struct Person {  
    char name[50];  
    char phoneNumber[20];  
    struct Address address;  
};
```

```
struct Person p1;
```

```
let p1 = {  
    name: "",  
    phoneNumber: "",  
    address: {},  
};
```

- In both these examples, we have created an *Object* p1.
- p1 contains three datum, name, phoneNumber, and address.
- We can create another Object, p2 which may contain a different name, phoneNumber, and address.
- In fact, we can create a whole collection of **Person** objects.

## 2.5 Objects: Encapsulating Behaviour

- Compound data types is a nifty extension to the built-in data types, we can now create our own types.

However:

- we still have to keep track of where we put the code to manipulate these data.
- all data is still available to anyone, we cannot protect access.

### Encapsulation

- Cleaner public interfaces of object/class/package/subsystem/system/service
- Protect data from unsanctioned access
- Focus each object/class on *their* task, remain unaware of other objects and tasks.
- *Easy To Change*

Example: How to get from *Victoria Station* to *Paddington Station*?

- Know every road and roadblock along the way?
- Know the bus-routes that will take you there?
- Know which tube-line to board?

- Know how to wave down a taxi?
- *Do not* know how to drive a car in London traffic.
- *Do not* know whether there is enough fuel in the taxi.
- *Do not* know how other travellers will get there, their names, or anything else about them.

## 2.6 Classes describe Objects

- Java and C++ are *typed languages*
- Every item of data must have a type
- At the very basic level it is used to allocate sufficient memory for the data item.
- The language enforces type consistency, if you'll let it.
  - (A **Person** can only be accessed as a person and not e.g. as a **Car** ).
- Developer-defined data type **Class**
- describes the structure of the data type
- gathers methods (functions) that operate on the data.
- Encapsulates methods and parts of the data type to create a cleaner interface.
- *Blueprint for creating objects*
- *Description for what a developer may do with an object*
- The programming language helps developers to “stick to the script”

## 2.7 One Class, many Objects

```
public class Person {
    private String name;
    private String phoneNumber;
    private Address address;

    // Constructors and methods
    // ...
}

// ...
Person p1 = new Person("Ada", "1", "Newstead Abbey");
Person byron = new Person("George Gordon", "0", "Newstead Abbey");
Person[] students = new Person[25];
System.out.println(new Person("Coleridge", "2", "Coleridge Cottage"));
```

- Same class is used to create many objects

- Each object contains the same attributes (variables), but with their own values.
- Each object contains the same set of methods (functions).
- Each object contains all the necessary data to describe *one and only one* instance of that type.
- You *may* have a variable that reference an object.
- The name of this variable is unknown to the object itself.
- The variable name can (and often will) change as you pass an object around.

## 2.8 Summary

- **We use objects to represent the real world.**
  - **Reduced cognitive gap (Real World Object  $\Leftrightarrow$  Object in Computer Program)**
- Classes describe Objects
- Classes, or Types, are necessary in programming languages such as Java or C++.
  - Encapsulates a compound data type
  - Encapsulates associated behaviour
- Object Oriented Programming with Classes is *one* language design choice
  - Other languages (e.g. JavaScript) do not *need* classes but they help readability and maintainability.
  - Other languages (e.g. Clojure) do not encapsulate behaviour together with data
    - \* focus instead on describing data types and their relations
    - \* made possible because data is immutable && code structure is enforced in other ways.

## 3 Compiled Languages

- Java and C++ are *compiled* languages
  - A tool translates your source code into something machine readable *before* you run it.
  - ... as opposed to e.g. Python, PHP, or JavaScript, where a tool *interprets* your source code at runtime.
- This adds a few steps to the toolchain, i.e. `write`  $\rightarrow$  `compile`  $\rightarrow$  `maybe-more-compilation`  $\rightarrow$  `run`

- If you use a modern IDE you probably do not have to worry too much about this.

#### Benefits

- Efficiency (performance *and* e.g. memory efficiency)
- Compiler will syntax-check *all* your code before you deploy
  - *s/runtime errors/compilation errors* – Fix your code before you deploy
  - Fosters a more stringent approach to programming
- Source code is not accessible to end-users
- Smaller size of shipped program
- Access to low-level APIs on your computer, e.g. Operating System, CPU, network, disk, memory, etc.

#### Challenges

- Compiled code *may* be platform dependent (C++ is, Java isn't)
- More complex toolchain
- Difficult to edit a running program on the fly
- Some programming language constructs are difficult to achieve, e.g. homoiconicity.
  - (But not impossible; Clojure accomplishes this)

### 3.1 Basic Steps

#### Java

1. Write Java Source Code
2. **Compile** to machine independent **bytecode**
3. **Interpret** bytecode

Optional: Package the bytecode files into a JAR file.  
C/C++

1. Write Source Code
2. **Compile** to **runnable binary** or **relocatable machine code**
3. **Link** relocatable machine code to a **runnable binary**
4. **Run** the **runnable binary**

## 3.2 Getting Started with Java

1. Make sure you have a JDK/JRE installed
2. Start a terminal
3. use `javac` to compile a `.java` file

- `javac Start.java`

4. use `java` to run a `.class` file

- `java Start`

Basic rules:

- Each class is defined in a file *with the same name as the class*
  - Class names are in principle case sensitive (lower/upper case letters)
  - In practice, some filesystems are not; this may create problems.
  - $\Rightarrow$  Use proper and unique names, and make sure the file and class are spelled the same way.
- The “root” Class, where you want the program to start *must* have a `main()` function:

```
public class Start {  
  
    public static void main(String [] args) {  
    }  
  
}
```

- It is good practice to keep this function *very* small; a simple printout and an object creation or two.

## 3.3 Getting Started with C++

1. Make sure you have a C++ compiler installed.
2. Start a terminal
3. Use your compiler (e.g. `g++`) to compile your `.cc` and `.hh` files.

- `g++ start.cc -o start`

4. Run the resulting program as usual:

- `./start`

Basic Rules:

- The compiler doesn’t care, but:
  - keep *class declaration* in a `.hh` - file
  - keep *class definition* in a `.cc` file with the same name as the `.hh` file.

- *somewhere* in the compiled program there must be a `main()` function.

```
int main() {
    // ...
    return 0;
}
```

- It is good practice to keep this function *very* small; a simple printout and an object creation or two.
- It is also good practice to keep this function in an easily recognisable file, e.g. `main.cc` or `start.cc`.

### 3.4 Build Tools (multi-file project)

- `javac` will follow class dependencies
  - *until* it encounters a class that does not need to be compiled (source code unchanged).
- C++ compiler will just do one file at the time; need to link everything together afterwards.
- Can use wildcards `javac *.java` to re-build everything.

Build tools save time

- Your IDE can help you (e.g. a project in Visual Studio)
- Be a Good Friend (TM), create a `makefile`.
  - <https://makefiletutorial.com/>

`VARIABLE = value`

`target: dependency`  
 Command to build target

### 3.5 Generic makefile for Java

```
SRC_DIR := src
OUT_DIR := out
DOC_DIR := doc
sources := $(wildcard $(SRC_DIR)/*.java)
classes := $(sources:$(SRC_DIR)/%.java=$(OUT_DIR)/%.class)

JC := javac
JCFLAGS := -d $(OUT_DIR)/ -cp $(SRC_DIR)/

.SUFFIXES: .java
.PHONY: all clean

all: $(classes)
```



```

$(classes): $(OUT_DIR)/%.class: $(SRC_DIR)/%.java
    $(JC) $(JCFLAGS) $<

doc: $(sources)
    javadoc -public -cp $(OUT_DIR) $(sources) -d $(DOC_DIR)

clean:
    $(RM) -r $(OUT_DIR)/*

run: all
    java -cp $(OUT_DIR) JavaPonies

```

### 3.6 Generic makefile for C++

```

CC = g++
CFLAGS = -g -Wall
INCLUDE = -I.
TARGET=myProgram
EXT = .cc
SRCS = $(wildcard *$(EXT))
OBJS = $(SRCS:$(EXT)=.o)

all: $(TARGET)

run: $(TARGET)
    ./$(TARGET)

$(TARGET): $(OBJS)
    $(CC) -o $@ $^ $(CFLAGS) $(INCLUDE) $(LDFLAGS) $(LIBS)

%.o : %$(EXT)
    $(CC) -c $< -o $@ $(CFLAGS) $(INCLUDE)

clean:
    $(RM) $(OBJS) $(TARGET)

```

- This will get you started, but the build file will need to be extended.
- Note that changes to `.hh` files will not be considered with this. Options:
  - Explicitly define `DEPS= file1.hh file2.hh` (bad idea)
  - Make sure you touch the right `.cc` - file (better idea, but may miss places where the file is included)
  - Plan your classes and APIs beforehand to minimise changes (even better idea)
  - Advanced `makefile-fu` to fix this once and for all with `g++ -M`.

## 4 Typade språk: Klasser och Objekt i Java

- All kod skrivs som en del av en klass
- (nästan) all kod körs som objekt.
- Varje objekt har en typ, varje *variabel* har en typ
- Varje objekt måste uttryckligen skapas : `new Pony()`
  - Gör plats i minnet för alla attribut som behövs för ett Pony-objekt
  - Kör Konstruktorn på objektet för att initiera det.
- Ett objekt kan “låtsas” vara av en annan typ
  - Om den implementerar ett eller flera **interface**
  - Om den ärver från en basclass

## 5 Datatyper

```
byte smallNumber = 127;
short largerNumber = 32767;
int normalNumber = 100000;
float smallDecimal = 0.123456f;
double largeDecimal = 0.12456789;
```

```
boolean trueOrFalse = false;
char singleCharacter = 'A';
String someText = "Longer, but not too long Text";
```

- Alla variabler måste deklareraras innan de kan användas.
- De inbyggda datatyperna har inga metoder, men fungerar med de inbyggda aritmetiska operationerna ( `+-*/%=` )
- Notera att `String` är en klass; så `String someText` skapar en referens till ett objekt.
- Java har också klasser som representerar de inbyggda datatyperna som objekt
  - e.g. `Integer.parseInt("123")`

## 6 Referenser till Objekt

- *Objektreferens* är en inbyggd datatyp i Java.
  - I C/C++ pratar man mycket om **pekare** och har ofta hela kapitel om pekararitmetik
  - I Java har man förenklat detta. Normalt är detta bra.
- Σ Alla variabler som hänvisar till ett objekt är en *referens*
- Att kopiera en variabel ( `a = b` ) kopierar referensen *men inte objektet*

## 7 Attribut, Parametrar, och Lokala Variabler

### 7.1 Attribut

- Attribut definieras i klassen
- Attribut har ett värde för varje objekt.
- Exempel `Car.myColour` ; Varje objekt av typen `Car` har sitt eget värde:
  - `c1.myColour == "red"`
  - `c2.myColour == "yellow"`
- Attribut kan definieras med ett startvärde
- Attribut kan förändras av metoder

### 7.2 Parametrar

- Parametrar definieras som en del av en metod
- Parametrar har ett nytt värde varje gång metoden anropas
  - Värdet anges av den anropande metoden.
  - e.g. `theCar.calculateFuelConsumption(theCar.getCurrentDistance(), 40)` // current distance in km, 40 litres
- Värdet på en parameter kan ändras inuti en metod, men det förändrar inte värdet hos den som anropar
- Parametrar kan ses som *Lokala variabler* där initial-värdet bestäms någon annanstans
- Parametrar *kan inte* anges med ett default-värde (i Java).

### 7.3 Lokala Variabler

- Lokala variabler kan deklarerarar var som helst inuti en metod.
- Lokala variabler kan bara användas *från den punkten och framåt*.
- Lokala variabler kan deklarerarar med ett startvärde.
- Lokala variabler kan ändras inuti en metod.
- Lokala variabler gäller bara inuti det block `{}` som de deklarerarar inom.

## 7.4 Exempel

```
public class Pony {
    private String name = "whoami?";           // "name" is an attribute

    public void setName(String theName) { // "theName" is a parameter
        int nameLength = theName.length;      // "nameLength" is a local variable

        if (0 < nameLength) {
            int internal;                      // "internal" is a local variable only available until th
            name = theName;
        }
        // "internal" is no longer accessible here
    }
}
```

## 8 Lokala Variabler och Objekt

```
public Pony createPony() {
    Pony thePony = new Pony("Twilight");
    thePony.setNewBehaviour();

    return thePony; // Vad returneras här?
}

public Pony modifyPony(Pony thePony) {
    thePony.setSpeak("We'll do everything by the book.");
    thePony.updateSpeakTimer(2000);
    thePony.speak();

    return thePony; // Vad returneras här? Är returen ens nödvändig?
}
```

## 9 Metoder

- Objekt har *metoder* (funktioner) som man kan anropa *på just det objektet*
- Ett litet mini-program som består av all data i det objektet och alla metoder i objektet.
- Metoder har **returvärde**, ett **namn**, och noll eller fler **parametrar**.
- Samma **namn** kan användas på fler metoder om de har olika **parametrar** (*polymorfism*)

Kompilatorn håller koll så att:

- Metoden du anropar finns för den objekt-typen
- Metoden finns med de parametrar du anger
- Försöker översätta parametrarna så att de passar

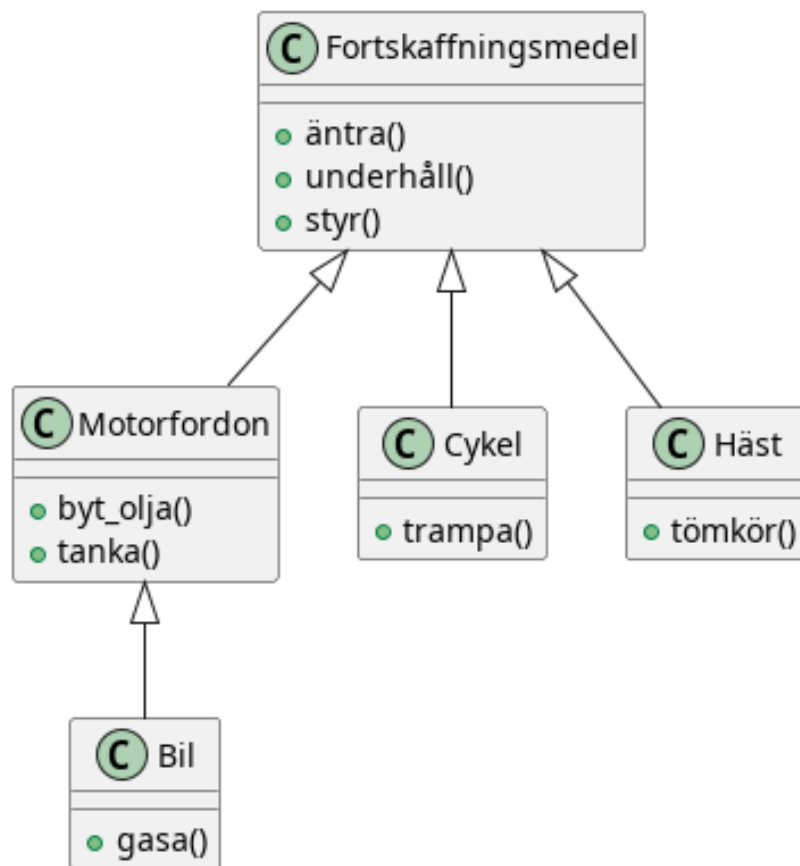
- Metoden returnerar ett värde av rätt typ
- Du sparar returvärdet i en variabel som får innehålla den typen.

## 10 Statiska metoder

- En *statisk metod* fungerar “på klassen” utan att man behöver ett objekt.
- En statisk metod kan bara läsa och skriva till statiska attribut.
- Vanligast är `public static void main(String [] args)`
- Också vanligt är varianter av `createInstance()`
- *Man bör undvika statiska metoder* om man inte har goda skäl (som ovan).
  - Det är nästan alltid bättre att först skapa ett objekt.
  - Statiska metoder saknar möjlighet att spara undan tillstånd, de har inget objekt att arbeta med.
  - Lättare att läsa, att använda vanliga metoder på objekt är normalfallet.
  - Lättare att hantera när man inser att man inte längre kommer undan med bara statiska metoder.

## 11 Arv och Interface

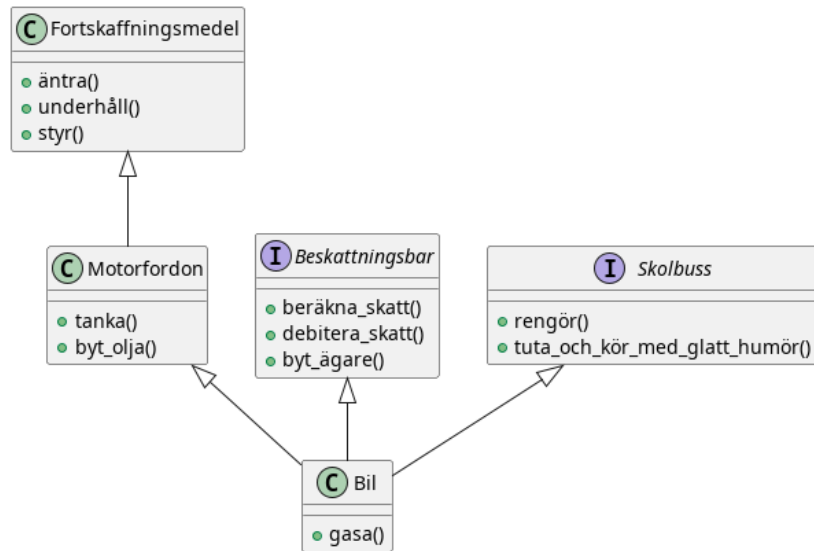
- Arv är viktigt i typade språk; det används för att kunna hantera ett objekt utifrån olika typer.
- Exempel:
  - En `Bil` är ett `Motorfordon` som är ett `Fortskaffningsmedel`
  - Om jag har ett objekt av typen `Bil` så vet jag att det *också* är ett `Motorfordon`, med alla metoder och attribut som deklarerats i den klassen.
  - Min samling av `Fortskaffningsmedel` kan innehålla objekt av olika typer, t.ex. `Bil`, `Cykel`, och `Häst`.
    - \* Jag kan använda alla metoder som deklarerats i klassen `Fortskaffningsmedel`
    - \* Jag kan *inte* använda metoder som deklarerats “längre ner” i arvshierarkin.
    - \* Jag *vet faktiskt inte längre* vilken faktisk typ något enskilt objekt i min samling har.



## 12 Arv och Interface i Java

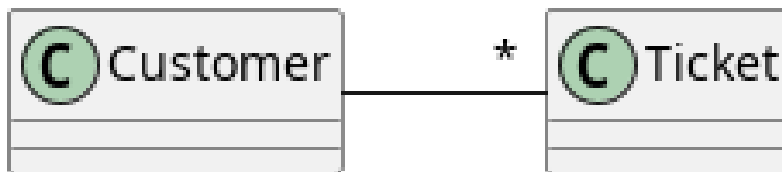
- Java har två typer av arv: **extends** och **implements**
  - **extends** *utökar* en existerande klass med nytt beteende (nya metoder) och nya attribut.
  - En klass får bara *utöka* en annan klass i Java (Multipelt arv är inte tillåtet).
  - **implements** implementerar metoderna som *deklarerats i ett interface*.
  - En klass får implementera hur många interface som helst.
- Extends använder man när man vill återanvända en viss implementation.
- Implements använder man för att ge en klass möjligheten att agera utifrån en viss *roll*
- Rollen gör att andra objekt kan förvänta sig att vissa metoder finns i objektet
- Exempel:
  - Hos skatteverket kanske min Bil också behöver vara **Beskattningsbar**

- I Taxiföretaget behöver objekt av typen *Bil* också vara *Skolbuss*-objekt



## 13 Collections

- Examples
  - A *collection of Customers*
  - A *collection of Movie Shows*
  - A Cinema has *many Seats*
  - A Customer has *one or several Tickets*
- Shown in Class Diagrams:



## 14 Fixed or Flexible Size

- *Sometimes* the collection has a fixed size
  - Or at least a fixed upper bound
- *Most of the time*, the size of a collection is not known at design time.

### Prefer Flexible Data Structures for collections

Examples of collection data structures:

**Vector** A flexible-size Array. Prefer `java.util.ArrayList`

**Set** Unordered collection where duplicates are not allowed.

**Bag** Unordered collection where duplicates *are* allowed.

**Queue** first in, first out

**Stack** Last in, first out

**Dictionary** Tuples of the form `<key, value>`

**Linked List** Mostly an internal implementation detail these days

**Tree** Can be quite useful for some problem types

**HashMap** A hash-value for each element decides where it is stored, makes searching fast

## 15 ArrayList

- Stores a collection of *objects*
- Is called a *generic class*
  - it can be instantiated with any type of objects
  - uses the *diamond notation* `<classname>`
  - once instantiated, it can only store elements of the type `<classname>`

```
import java.util.ArrayList;
```

```
ArrayList<String> myList = new ArrayList<>(); // Element Type is inferred from the variable  
myList.add("Hello");  
myList.add(new String("World"));  
System.out.println(myList);
```

```
// Built-in Datatypes do not work  
// ArrayList<int> myIntList = new ArrayList<int>();  
ArrayList<Integer> myIntList = new ArrayList<>();  
myIntList.add(Integer.valueOf(42));  
myIntList.add(12); // 12 can be "upgraded" to an instance of the Integer class  
System.out.println(myIntList);  
System.out.println(myIntList.get(0) instanceof Integer);
```

## 16 Index number in Collection

- Elements in an ArrayList range from `[0 \dots size()-1]`
- When accessing an element, *check* that the index is within this range.
- Adding or removing an element in the middle reorders every element after.
- Accessing elements by index *may* be useful



– Personally, I prefer not to if I can avoid it.

```
if (0 <= ticketNumber    // We start at index 0
    && myTickets.size() > ticketNumber) { // size() is just outside of the collection.
    String details = myTickets.get(ticketNumber).toString();
    System.out.println(details);
}
```

## 17 Traversing a Collection: for-each

```
Customer c = new Customer();
c.addTicket(new Ticket("Spartacus", "19:00 tonight"));
c.addTicket(new Ticket("Ben Hur", "15:00 this afternoon"));

for (Ticket t : c.getTickets()) { // For each element t of the type Ticket in collection c
    System.out.println(t.toString());
}
```

## 18 Filtering a Collection

```
Customer c = new Customer();
c.addTicket(new Ticket("Spartacus", "19:00 tonight"));
c.addTicket(new Ticket("Ben Hur", "15:00 this afternoon"));

for (Ticket t : c.getTickets()) {
    if (t.getName().contains("tonight")) {
        System.out.println(t.toString());
    }
}
```

- Filtering collections is *extremely* useful
- In functional programming, it is usually the starting point of nearly *everything*:
  1. Given a collection
  2. Filter to remove everything not relevant
  3. Do something with the remaining elements
  4. (maybe) repeat steps 2 and 3
  5. Translate whatever remains to the format you want
  6. ...
  7. Success!
- Many languages have built-in support for this.
- Later versions of Java support it with the **Streams** API (which we will not cover in this course).

## 19 Other forms of Iteration: while

```
int x = 5;

while (0 <= x) {
    System.out.print(" " + x);
    x--; // If you forget this line, x will never update and the while loop will continue fo
}

System.out.println();
System.out.println("x = "+x);
```

- Repeat while some condition tests to **true**
- Can go on forever, if you are not careful
- Often used if you do not know when to end, e.g.
  - *while (user has not exited the menu)*
  - *while (there are more elements in the database)*
  - *while (there are more lines in this file)*
  - *while (I still have not found a movie that shows tonight)*
- Boolean expression can be arbitrarily complex: `while (index < myTickets.size() && !found && !userAborted)`

## 20 While without index

```
int f0=0;
int f1=1;
int fn=f1 + f0; // Fibonacci Sequence

while (fn < 100) {
    System.out.print(" " + fn);
    f0 = f1;
    f1 = fn;
    fn = f1 + f0;
}
```

## 21 Arrays

```
int[] someNumbers = new int[5];
someNumbers[0] = 42;
someNumbers[3] = 12;

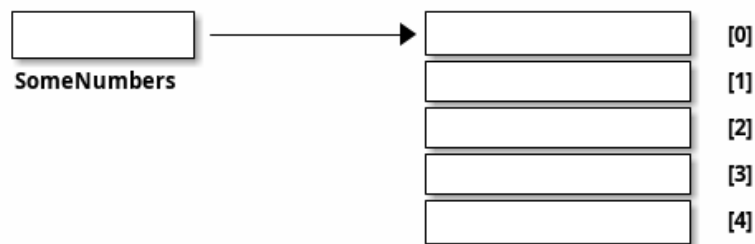
for(int num : someNumbers) {
    System.out.print(" " + num);
}
```

```
System.out.println();
int [] otherNumbers = {23, 31, 57};
System.out.println(otherNumbers.length); // Note -- length is an attribute and not a method
```

- Arrays are built-in and can operate on any type.
- Arrays are *fixed size*, extending or shrinking has to be implemented by yourself
- Inserting and removing elements has to be implemented by yourself
- *May* have better performance than e.g. ArrayList

## 22 Deeper into Array Creation

```
int [] someNumbers;
someNumbers = new int[5];
```



- `int [] someNumbers` creates a variable that holds a reference to an array
- `new int [5]` allocates *consecutive* space for 5 integers.

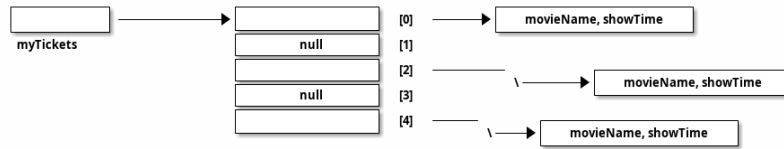
## 23 Array of Objects

- What happens if we allocate an array of e.g. Tickets?
- What is the output of:

```
Ticket [] myTickets = new Ticket[5];

for(Ticket t : myTickets) {
    System.out.println(t.toString());
}
```

## 23.1 Objects and Object References



## 24 Yet another iteration: for

```
for (int i = 0; i < 10; i++) {
    System.out.print(" " + i);
}
```

- `for (<initialisation> / ; <condition> / ; <increment>) { <statements> }`
- Difference to for-each is that we can use the iterator (e.g. `i` above) inside the loop
- Does not have to look like above, initialisation, condition, and increment can be quite different:

```
for (Query q=new Query("Select * from Users") ; q.hasMoreElements(); q.nextElement() ) {
    System.out.println(q.currentElement());
}
```

```
for (Iterator<Ticket> it = myTickets.iterator() ; it.hasNext() ; /* empty increment */ ) {
    Ticket t = it.next();
    // ...
}
```

-Compare to the while loop:

```
<initialisation>;
while (<condition>) {
    <statements>
    <increment>
}
```

## 25 Paket

- I Java strukturerar man sin applikation med hjälp av *Paket*:
  1. En katalog i filsystemet: `src/model`
  2. En deklaration *överst* i de .java-filer som ingår i paketet: `package model;`
- Man kommer åt de paket som finns i `$CLASSPATH`

- Måste deklarera att man vill använda en viss klass från ett annat paket:  
`import package.name.Class;`
- Man kan vara lat och importera alla klasser i paketet: `import msv.example.*`

## 26 Sammanfattning

- Java är ett *kompilerat* språk
  - Kompilatorn översätter från din skrivna kod till ett körbart program
  - Kompilatorn kontrollerar att allt är rätt och allt hänger samman
    - \* (Det finns såklart fortfarande fel som kan uppstå under körning)
- Java är ett typat språk: Alla variabler har en viss typ
  - Kompilatorn kontrollerar att du använder varje variabel på rätt sätt.
- Ett java-program är uppbyggt som en samling *paket*, där varje paket innehåller olika klasser.
- (nästan) allt är objekt i Java, och varje objekt har en *klass* som beskriver den.
- Man bygger sitt program genom att ha objekt som samarbetar för att lösa varje uppgift
  - objekten har referenser till andra objekt
  - objekten anropar metoder på andra objekt
- Vill man att ett objekt skall kunna låtas vara flera typer så använder man sig av *arv*
- Olika sorters *Samlingar* av objekt: Vector, Array, List, ArrayList, Tree, Bag, Dictionary, ...
- Många program handlar om att hantera samlingar av objekt på olika vis eller att upprepa beteende
  - Iterera över samling: *for-each*
  - Upprepa beteende: *for* eller *while*

## 27 Nästa Föreläsning


- Användargränssnitt i Java
  - textbaserade gränssnitt
  - Grafiska gränssnitt med Java AWT/Swing
- Enkel filhantering i Java
- En introduktionsserie om Java Swing på Youtube: [https://youtube.com/playlist?list=PL3bGLnkkGnuV6991P\\_f9DvxyK51MFpq6U&si=q1T41nIORJXuEB\\_p](https://youtube.com/playlist?list=PL3bGLnkkGnuV6991P_f9DvxyK51MFpq6U&si=q1T41nIORJXuEB_p)

## 28 Övning: Mera Ponies

- Vi fortsätter med JavaPonies.
- MLP-fansen har beställt ett utökat gränssnitt där man kan få reda på mer data om varje Ponny.
- Vi skall dessutom se till att påbörja implementationen av *Interactions*, att en ponny byter beteende för att de är nära någon annan.

### 28.1 MLP-Data

1. Skriv ett interface `src/model/PonyStatistics.java` enligt nedan.
2. Se till att `model.Pony` implementerar detta interface. Notera att
  - Några metoder redan finns, men kan behöva utökas
  - Några metoder kommer kanske anropas flera gånger; särskilt `load()` kommer behöva ta hänsyn till detta.
  - Nya klasser kan behöva skapas t.ex. för att innehålla en *Interaction*
  - Några av get-metoderna kan behöva iterera över en `ArrayList<>` av t.ex. *Behaviours* för att plocka fram deras namn och spara i en *String*-array.
  - Vi inte har något sätt att använda dessa metoder ännu. *Skriv gärna enhetstester i stället.*

 <i>PonyStatistics</i>
<ul style="list-style-type: none"><li>● <code>void load()</code></li><li>● <code>String getName()</code></li><li>● <code>Path getDefaultImagePath()</code></li><li>● <code>String[] getCategories()</code></li><li>● <code>String[] getBehaviourGroups()</code></li><li>● <code>String[] getBehaviourNames()</code></li><li>● <code>String[] getEffectNames()</code></li><li>● <code>String[] getSpeakLines()</code></li><li>● <code>String[] getInteractionNames()</code></li></ul>

## 28.2 Interactions

- För att en Ponyy skall kunna interagera med en annan Ponyy, så krävs det att de vet att de står i närheten av varandra.
- Detta kan inte en enskild Ponyy veta. Vilka andra alternativ har vi?
  - `view.PonyWindow` vet var en viss Ponyy är (genom att fråga den), men inte de andra.
  - `view.MainWindow` har, när den skapat `PonyCard` för varje Ponyy, inte ens koll på vilka Ponnys som finns.
  - `model.PonyContainer` kan veta. Men då får den två ansvarsområden: Underhålla samlingen av Ponnys *och* sköta interaktioner.

Σ

1. Vi behöver skapa en ny klass `model.InteractionManager`, som har en samling med `InteractionEntity`
  - Metoden `maybeStartInteraction()` behöver anropas regelbundet
2. Vi behöver skapa ett interface `model.InteractionEntity` som Pony implementerar
  - Särskilt viktig är metoden `maybeStartInteraction()`
3. Vi behöver skapa en klass `model.PonyInteraction` som representerar en specifik möjlig interaktion.
4. Klassen `JavaPonies` behöver “sätta igång” ett `InteractionManager` - objekt.
5. Klassen `view.PonyWindow` behöver samarbeta med `model.Pony` så att `model.Pony` vet om den är synlig eller inte.

