Object orientation

Even more

Tobias Andersson Gidlund

tobias.andersson.gidlund@lnu.se

Agenda

Introduction

- This lecture will look at additional object oriented concepts.
- It is primarily different types of *relations* between classes.
  - How they communicate with each others.
- A class diagram will be shown to help you understand the end goal.
- To help you we will also give you some tips and tricks for finding classes.
- The lecture ends with a discussion on a number of existing classes.
  - Among others `BigInteger` and `StringBuilder`.

Relations

What is a relation?

- A relation is a connection between two different modelling elements.
  - Here mainly between different classes *or* objects.
- This lecture will cover:
  - *Links* between objects
  - *Associations* between classes:
    - Pure associations
    - Aggregate
    - Composition (aggregate composition)

What is a link?

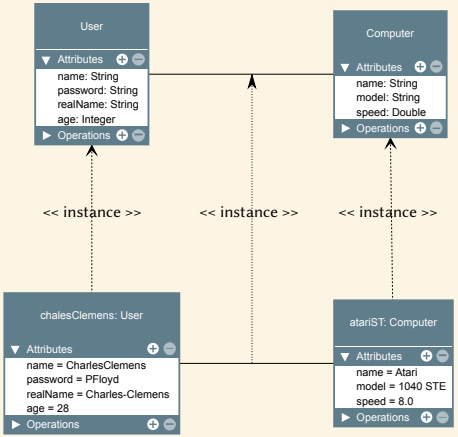
- A link is a connection between objects.
  - That is *not* between classes.
- A link is how the objects communicate.
- Objects send messages to each other via links.
- A message means that an operation is called.
  - This is what makes an object *do* something.
- In UML a link can exist between two, and only two, objects.
  - Shown with a line.
  - Can be drawn in two ways, either separately or in line.

What is an association?

- An association is a relation between classes.
  - Also called “is-a” relation.
- In the same way a link is between objects, associations are between classes.
- An association therefore means that there is a link between the objects that are instantiated from the classes.
- A link is an instantiation of an association.
- Notice that normally you never show both links and associations in the same diagram as they show different things.
  - On the next page you will, however, see such a diagram to visualise what it means.

Example

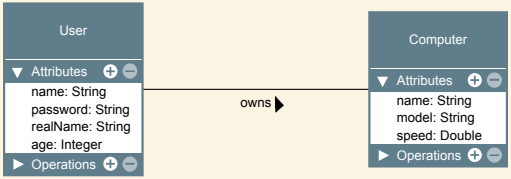
- Below is an example of an instantiation.

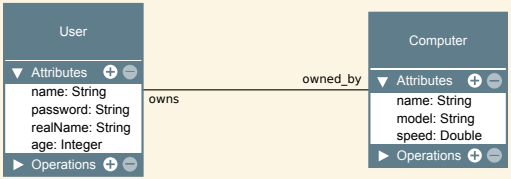


How associations can be drawn

- Associations are drawn as a line between classes.
- An association can have a *role name* or an *association name* – but never both at a time.
  - A role name says what role the class has in the relation.
  - An association name says what the entire relation means.
  - Use clear and descriptive names.
- To clarify the how to read it, it is possible to specify the reading direction with a little black triangle.
  - Notice that there is a difference between how to *read* the association and how to *navigate* it – more on that later.

Example


User
Attributes
name: String
password: String
realName: String
age: Integer
Operations
owns
Computer
Attributes
name: String
model: String
speed: Double
Operations

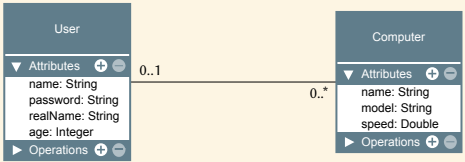

User
Attributes
name: String
password: String
realName: String
age: Integer
Operations
owns
owned_by
Computer
Attributes
name: String
model: String
speed: Double
Operations

Multiplicity

- An important property for a relation as an association is to say *how many* objects participate.
- This is called the *multiplicity* of the association.
- Multiplicity is marked at both classes participating in the relation.
- When multiplicity is read, it is done from the individual object on one side and the number of objects on the other side.
  - In a sense you “jump” the multiplicity on your own side.
- The number is written by a number or a range alternatively a star for “many”.

Example

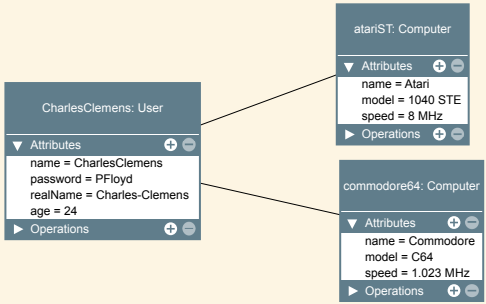
- The following association is read:
  Each user can have zero or more computers.
  Each *computer can have zero or one user.*



- Notice that when you read from the side with the greater multiplicity, you still read “each” and *not* “every”.

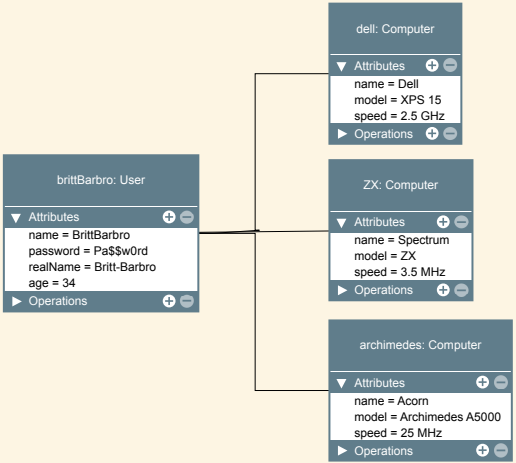
Example of instantiation

- As previously, one user having two computers. Som tidigare, en användare har två datorer.



- A system can have many users and computers *at the same time*.
  - Instantiation shows what *actual* connections there are at *a given time* in the system.

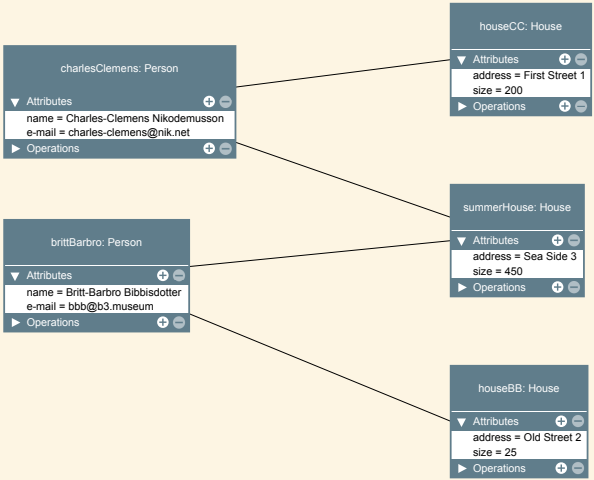
Additional example



Different multiplicity

- If no multiplicity is stated, it is *undefined*, that is, there is no “standard multiplicity”.
- The most common is to define a “one-to-many” relation.
  - If the relation is “one-to-one” it is often just a part of the other class.
- Sometimes a “many-to-many” relation needs to be done.
- The following is an example that will be instantiated in the next slide.

Example instantiation



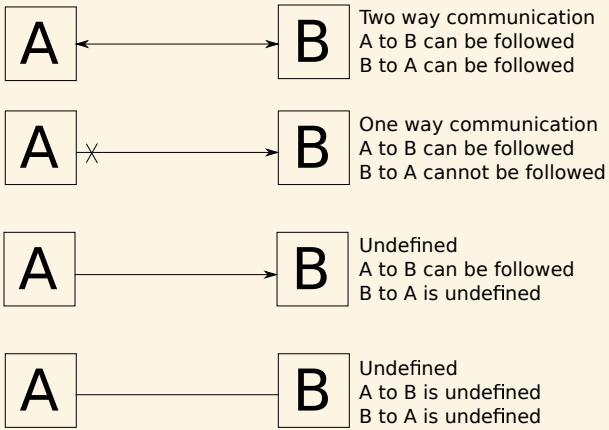
Navigability

- Navigability states if it is possible to “go” from one object to the other.
  - That is, in what direction the messages can be sent.
- Previous examples have all been bidirectional, but that is not always what you want.
- It is also a lot more difficult to *implement* two way communication.
  - Not important in the analysis phase, but important during design.
- It might also be possible to “go round”, that is get to an object by sending messages through other objects.
  - The cost for that in the implementation is high so it should be avoided.

Example navigability

- In the following example it is only possible to go from **Order** to **Product**.
- The semantics is that an order stores information on what set of products it contains, but each individual product does not know what order it belongs to.
- In UML it is possible to use a cross to say that it is not possible to navigate to one side.

Summary of navigability



Practice

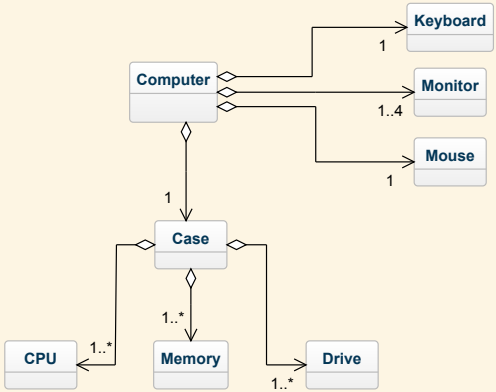
- Navigability as described here was introduced in UML 2.x.
- Too much navigability makes the diagrams hard to understand.
- Therefore, practice is:
  - Only show navigability when it is important.
  - Avoid the cross.
- In real life, it is understood that:
  - Association with an arrow is only possible to follow in that direction.
  - Associations without arrows are seen as two way communication.
- The only implication of this is that it is impossible to show non-followble associations, but what good are they...

Aggregation and Aggregate Composition

- Both aggregate and aggregate composition are stronger types of association.
- Instead of only being associated with the other class, they are *part of* each other.
- For aggregate the parts can exist independently of the whole.
- In UML an aggregate is shown as an open diamond on the “whole” part.
- Other than that, all other rules for association still apply.

Example

- A computer consists of different parts that all can exist without the whole.

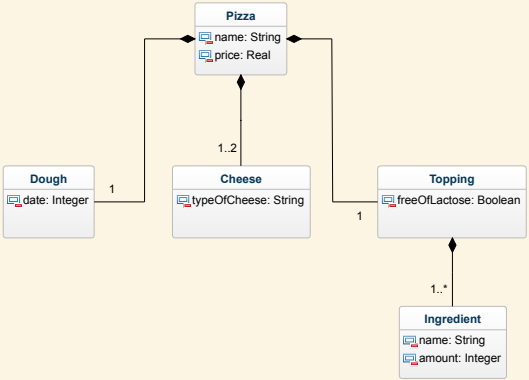


Aggregate composition

- An aggregate composition is a stronger whole-part-relation.
- A class “owns” the other parts.
  - If the whole is copied or removed, the parts are also removed.
- In UML the aggregate composition is shown as a filled diamond.
- In contrast to the pure aggregate, the multiplicity in an aggregate composition on the “whole” can only be 0 or 1.
  - A part can only be part of one whole.

Example

- A pizza consists of different parts and when the pizza is done the parts cannot be removed from the whole.



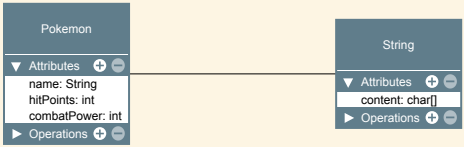
Relations in Java

Relations

- As stated, there are several types of relations in objectorientation.
- For Java there are primarily “just” two – associations and inheritance.
  - Not too strange as aggregate and aggregate composition only are degrees of association.
  - Inheritance is covered in the next course and not at all here.
- For Java we will see several variants of *knows-of* (association) implementation.
  - The difference is most often the multiplicity.

One-to-one relation

- The simplest case is if the classes have a *one-to-one* relation.
- This is actually something we have already done...



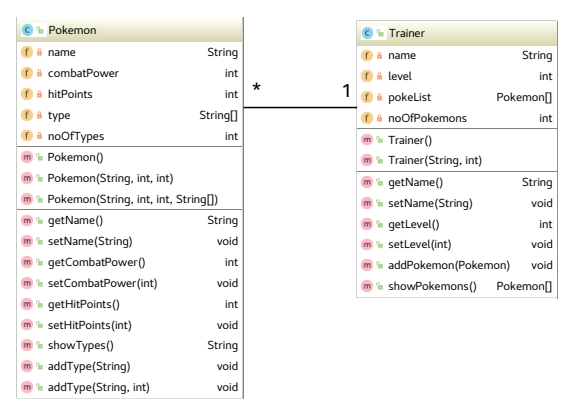
- This kind or relation is seldom modelled as it is so strong.

One-to-many relations

- If a relation knows of many other objects an array is most often used.
  - Or an `ArrayList` or any other structure you will get to know later.
  - An example of this relation can be that a `Person` can own several `Car`s.
- In programming it is also important to define navigability, try to keep it as simple as possible.
  - In the case of `Person` and `Car` it is much simpler if a person can own several cars, but each car does not know its owner.
- The book additionally shows how aggregation and composition works, which are only stronger forms of association.
  - In practice it will be implemented similarily.

Example

- A Pokémon trainer can have several Pokémon, but each Pokémon can only have one trainer.



Trainer class

```
package lecture7;

import java.util.Arrays;

public class Trainer {
    private String name;
    private int level;
    private Pokemon[] pokeList=new Pokemon[10];
    private int noOfPokemons = 0;

    public Trainer() {
    }
    public Trainer(String name, int level) {
        this.name = name;
        this.level = level;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getLevel() {
        return level;
    }
    public void setLevel(int level) {
        this.level = level;
    }

    public void addPokemon(Pokemon newPokemon) {
        if(noOfPokemons < 10) {
            pokeList[noOfPokemons] = newPokemon;
            noOfPokemons++;
        }
    }

    public Pokemon[] showPokemons() { // Deep copy of array!
        Pokemon[] safeArray = new Pokemon[noOfPokemons];
        for(int i = 0; i < noOfPokemons; i++) {
            safeArray[i] = new Pokemon(pokeList[i].getName(),
                    pokeList[i].getCombatPower(),
                    pokeList[i].getHitPoints());
        }
        return safeArray;
    }
}
```

Main program

```
public class PokeTrainer {
    public static void main(String[] args) {
        Trainer ash = new Trainer("Ash Ketchum", 40);
        Pokemon pikachu = new Pokemon("Pikachu", 286, 98, new String[] {"Electric"});
        Pokemon snorlax = new Pokemon("Snorlax", 3020, 180, new String[] {"Normal"});
        Pokemon slowpoke = new Pokemon("Slowpoke", 1024, 98, new String[] {"Water", "Psychic"});

        ash.addPokemon(pikachu);
        ash.addPokemon(snorlax);
        ash.addPokemon(slowpoke);

        System.out.println("Trainer " + ash.getName() + " has the following Pokémon:");

        for (Pokemon p: ash.showPokemons()) {
            System.out.println(p.getName());
        }
    }
}
```

Utdata:

```
Trainer Ash Ketchum has the following Pokémon:
Pikachu
Snorlax
Slowpoke
```

A note on ”Deep copy”

- The method for showing the Pokémon we *copy* all the values to a new array.
- In Java there is a method called `Arrays.copyOfRange()` that seems to do what we want.
- The problem is that it does a *copy-by-reference*, that is the *references* are copied, not the objects.
- If the above, or similar, built in method is used, it is still possible to change the original values in the array.

```
Pokemon[] pokes = ash.showPokemons();
pokes[1].setName("Lovecraft");
```

- This breaks encapsulation.

Navigability

- In the example it is only the trainer how knows of his Pokémon, but each Pokémon does not know how the trainer is.
- To make this possible, there must exist an attribute in Pokémon that is of type `Trainer`.

```
private Trainer trainer;
```

- A method to set such a value must also exist, and also one to show the name of the trainer (or, if you like, the complete object).

```
public String showTrainerName() {
    return trainer.getName();
}

public void setTrainer(Trainer trainer) {
    this.trainer = trainer;
}
```

In the training class

- The training class also needs to be updated with a call to the `setTrainer()` method for every added Pokémon.
- In this class, the current trainer must also be sent to the Pokémon object by sending `this`.

```
public void addPokemon(Pokemon newPokemon) {
    if(noOfPokemons < 10) {
        pokeList[noOfPokemons] = newPokemon;
        noOfPokemons++;
        newPokemon.setTrainer(this); // New!
    }
}
```

- In this way the two objects are connected together.
- Notice that it is still possible to call `setTrainer()` outside the `Trainer` class.
  - Can lead to problems if the Pokémon still is part of the list – inconsistency.

Example main program

```
public class PokeTrainerPoke {
    public static void main(String[] args) {
        Trainer ash = new Trainer("Ash Ketchum", 40);
        Pokemon cubone = new Pokemon("Cubone", 87, 38, new String[] {"Ground"});

        ash.addPokemon(cubone);

        System.out.println("Trainer for " + cubone.getName() + " is " + cubone.showTrainerName());

        System.out.println("Trainer " + ash.getName() + " has the following Pokémon:");

        for (Pokemon p: ash.showPokemons()) {
            System.out.println(p.getName());
        }
    }
}
```

Output:

```
Trainer for Cubone is Ash Ketchum
Trainer Ash Ketchum has the following Pokémon:
Cubone
```

Somewhat safer...

- If you want even safer code then the Pokémon class needs to check the list for the `Trainer` to see if it is in it.
- In this case we get the list from the Pokémon class to see if is in it (by checking content).
- Bara då är det okej att lägga till tränaren.

```
public void setTrainer(Trainer trainer) {
    Pokemon[] listOfPokemon = trainer.showPokemons();

    boolean found = false;
    for(Pokemon p: listOfPokemon) {
        if(p.getName() == this.getName() && p.getCombatPower() == this.getCombatPower()
        ↪  && p.getHitPoints() == this.getHitPoints()) {
            found = true;
            this.trainer = trainer;
        }
    }

    if(!found) {
```

Main program

- Below is an example of a main program.
- It is now starting to be a bit difficult to manage, so it is better to rethink how safe the program needs to be.

```
public class PokeTrainerPokeSafer {
    public static void main(String[] args) {
        Trainer ash = new Trainer("Ash Ketchum", 40);
        Pokemon pidgey = new Pokemon("Pidgey", 27, 12, new String[] {"Normal", "Flying"});
        ash.addPokemon(pidgey);

        System.out.println(pidgey.showTrainerName());

        Pokemon cubone = new Pokemon("Cubone", 87, 38, new String[] {"Ground"});
        cubone.setTrainer(ash);
    }
}
```

Output:

```
Ash Ketchum
Sorry, not possible
```

Relations

- It is important to implement relations between classes.
- All messages sent between classes are done over this relations.
- In programming it is easiest to create one way relations, but sometimes two way is needed.
- Think of encapsulation, but also on what is practical to work with.
- Next step will be more about object thinking – how to find classes.

Finding Classes

A good class

- The name must clearly reflect the purpose of the class.
- The class must be a clear abstraction from the problem domain.
- It must be clear what task(s) from the problem domain it manages.
- It must have high coheision.
  - Must only model one (and only one) abstraction.
  - The responsibilities of the class must be semanticially related.
- It must have low coupling.
  - It must *only* be connected to as many other classes as it needs.

Guidelines

- Three to five responsibilities per class.
- Each class needs to collaborate with other classes.
- Watch out for small classes.
- Watch out for large classes.
- Beware of “functoider” – classes with only technical functionality.
- Beware of “omnipontent” classes – “god classes” doing everything.
- Later, when inheritance is covered, beware of too deep hierarchies.

How to find classes

- Perform noun/verb analysis on the documentation.
  - Nouns are candidates for classes.
  - Verbs are candidates for responsibilities (operations).
- An idea is to use CRC analysis cards.
  - Cards with class name, responsibilities and collaboration.
- With all techniques, beware of:
  - Synonyms – words with the same meaning.
  - Homonyms – words with different meaning but looking the same.
- Look for “hidden classes”.
  - Classes not clearly visible from the description.

Noun/verb analysis

- Collect all relevant documenation.
  - Descriptions of the business
  - Requirements analysis
  - Use cases
  - Everything else!
- Create a list of nouns and noun phrases.
- Create a list of verbs and verb phrases.
- Try to connect responsibilities (verbs) with candidate classes (nouns).
- Add attributes to be able to handle the responsibilities.

Class diagram

- The end result will be a *class diagram*.
- This diagram will give a static overview of how the system will work.
  - What parts it consists of.
  - How it communicates.
  - What information it contains.
- In UML this is naturally complemented with dynamic diagrams to see what happens in the system over time.
- The diagrams, with the class diagram most prominent, gives the developer a tool to use to understand the system to be developed.

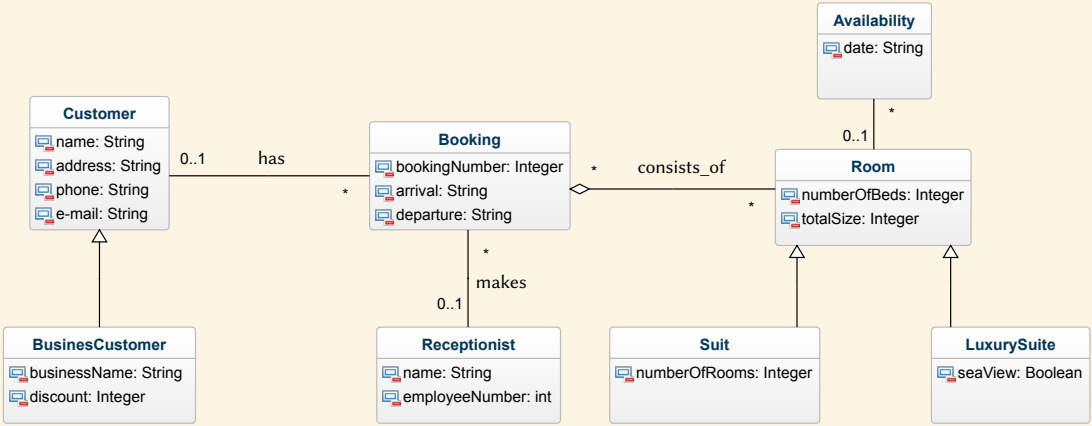
Example

- The following is a very small example based on this business description:

  A small family owned hotel has twenty two rooms and they want a booking system. The customers call and book one or more rooms via a receptionist that is able to see the availability of the rooms. It is important to connect the receptionist with the booking if something goes wrong. In addition the hotel wants contact information for the customers including address and e-mail. If it is a business customer this also needs stored in the system and the hotel holds an informal list of discounts for their favourite business customers. The rooms are either standard rooms with a number of beds (from one to four) or a suite (three of those) alternatively a luxury suite (two available, one with view over the sea).

- The next slide shows a possible class diagram for this business.

Example diagram



Classes In The API

Primitive types as objects

- In Java there are eight different primitive data types and for each there is a corresponding class.
  - Most often it is the name of the primitive type but with capital first letter.
  - For example `Float` and `Double` but also `Integer` and `Character`.
- The primitive data types are most often more efficient, but several methods in the API will need objects and not primitive types.
  - The class version is also useful when creating generic classes, but more on that in a later course.
- For each class there are also a number of methods that makes it easier to calculate, convert and much more.

Using the class version

- It is important to notice that the class version is *immutable*.
  - When it has a value, it cannot be changed (but it can be overwritten).
- They need to get a value when they are created, there is no empty constructor.

```
Integer theTruth = new Integer(42);
```

- Also notice that other data types, like `String`, can be used as parameter and will be automatically converted.

Boxing

- To make working with both primitive and class versions at the same time there is *boxing*.
- This mechanism packages or unwraps the values to its counterpart.

```
public class PrimitiveClasses {
    public static void main(String[] args) {
        Integer truth = new Integer(42);
        Integer c64mem = new Integer("64");

        System.out.println(truth + c64mem);

        Integer c128mem = 128; // autoboxing

        System.out.println(c128mem);
    }
}
```

`BigInteger` and `BigDecimal`

- To be able to handle really large numbers there are two classes:
  - `BigInteger` can, in theory, hold almost any integer.
  - `BigDecimal` has all the decimals correct.
- Notice that both classes are *much less efficient* than the primitive data types.
- Just as the previous classes they are *immutable*.
- Both classes takes *strings* as parameters as larger numbers than can be printed by an `int` or `double` can be used.
- They also need to use their own methods for arithmetics rather than the usual symbols.

Example

```java
package lecture7;

import java.math.BigInteger;

public class LargeNumbers {
    public static void main(String[] args) {
        BigInteger large = new BigInteger(String.valueOf(Long.MAX_VALUE));
        large = large.add(new BigInteger("10"));

        System.out.println("Largest integer is: " + Long.MAX_VALUE);
        System.out.println("Even larger is:     " + large);
    }
}
```

Output:

```
Largest integer is: 9223372036854775807
Even larger is:     9223372036854775817
```

The class `StringBuilder`

- Just as previous classes a normal `String` is immutable.
- As modifying strings is something often desired, there is a class that allows for that – `StringBuilder`.
- `StringBuilder` has many more methods than an ordinary `String` has.
  - Some of them return a new `StringBuilder`, so be careful to check this.
- `StringBuilder` is less efficient than an ordinary `String` so if no changes are necessary it is better to use the simpler type.

Example

```
public class StrangByggare {
    public static void main(String[] args) {
        StringBuilder str = new StringBuilder("Gotta Catch 'Em All");
        System.out.println(str);

        str.append('!');

        System.out.println(str);

        str.replace(6, 11, "Kill");

        System.out.println(str);
    }
}
```

Utdata:

```
Gotta Catch 'Em All
Gotta Catch 'Em All!
Gotta Kill 'Em All!
```

Summary

- These two lectures have been an introduktion to the most fundamental parts of object orientation in Java.
- As stated previously OO is more of a *mindset* rather than a notation – you need to practice.
- Next lecture will be an exercise on modelling.
- Bring pen and paper!
- Notice that one of the most important parts of OO has not been covered yet – inheritance.
  - More on this in the next course.