Heute

- Interface *Map*
- Algorithmisches Denken
- Klassenentwurf
- Projekt Scheduling

Java *interfaces*

- *Interfaces* sind eine besondere Form von abstrakten Klassen:
  - Alle Methoden sind *public* und *abstract*.
  - Es gibt keine Konstruktoren.
  - Es gibt keine Attribute außer Konstantendefinitionen, die *public*, *static* und *final* sind.

- D.h., dass eine nicht-abstrakte Unterklasse eines Interfaces **alle** Methoden des Interfaces implementieren muss.

- Insofern ist ein Interface eine Spezifikation, die angibt, über welche Methoden ein Objekt dieses Typs verfügt.

Interface *Map* (in java.util)

- Maps speichern *Schlüssel-Wert* (key-value) Paare.
  - *Map<K,V> aMap = new HashMap<K,V>();*
  - *aMap.put(key,value);*
  - *V value = aMap.get(key);*
  - *aMap.remove(key);*

- Implementierte Klasse:
  - HashMap
    - verwendet hashCode() der Keys zum Abspeichern und Suchen

Interface *Map* (in java.util)

Methoden in Map:

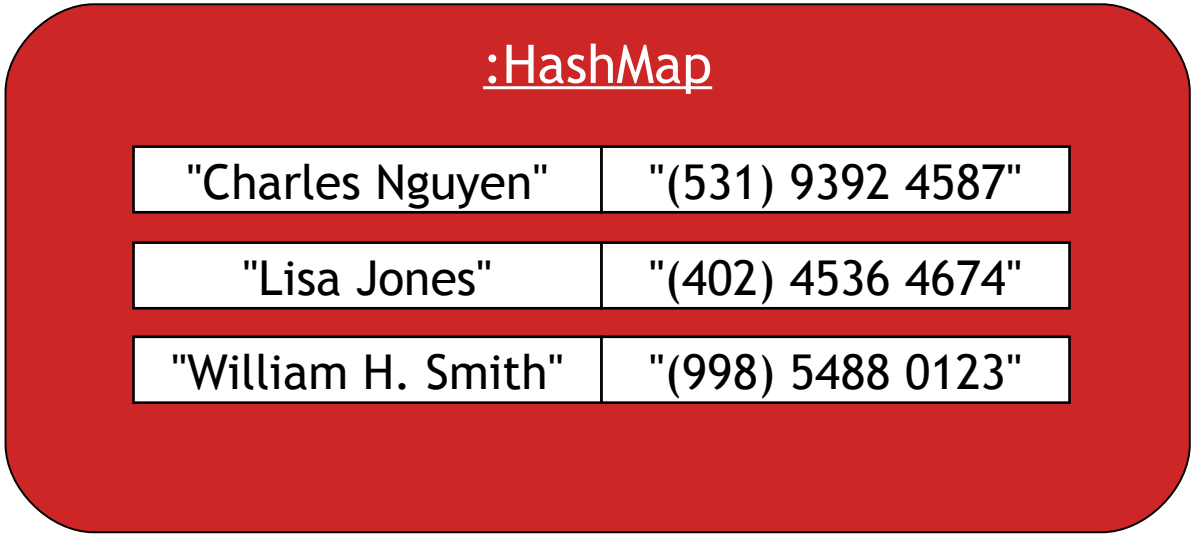
- *aMap.put(key,value)*
  - fügt neues Paar *(key, value)* hinzu (falls *key* nicht in Map gespeichert)
  - Ersetzt bestehendes Paar (falls *key* schon vorkommt)
- *aMap.get(key)*
  - gibt entsprechendes *value* zurück bzw. *null*, falls *key* nicht in Map vorkommt.
- *aMap.remove(key)*
  - löscht Paar mit *key* (falls vorhanden)

Beispiel

```
HashMap <String, String> phoneBook =
                new HashMap<String, String>();

phoneBook.put("Charles Nguyen", "(531) 9392 4587");

String phoneNumber = phoneBook.get("Lisa Jones");
```



Umgang mit Map

- Auf das Schlüsselset einer Map kann mit Methode

  keySet()

  zugegriffen werden.
- Wenn man bestimmten Schlüssel *key* sucht, ist ein

  get(key)

  einfacher als eine Schleife über das KeySet!

Umgang mit Map

get(k)

gibt *null* zurück, wenn Key *k* nicht vorkommt.

remove(k)

hat keine Wirkung, wenn Key *k* nicht vorkommt.

Umgang mit Map

- Mit

  put(k,v)

  wird ein bestehender Wert für den Schlüssel *k* in der Map überschrieben.

- Ein *remove(k)* ist nicht nötig!

Projekt Scheduling

- Haben Liste von Aufträgen unterschiedlicher Bearbeitungslänge, die möglichst gleichmäßig auf zwei Anlagen verteilt werden sollen.

Entwurf von Klassen

- Wie sollen die Klassen für eine Problemstellung gewählt werden, sodass die Lösung leicht zu verstehen und zu verändern ist?
- Merkmale für die Qualität der Lösung:
  - Geringes Coupling
  - Hohe Cohesion
  - Wenig/keine Code duplication
- Vorgehen:
  - Responsibility-driven design
  - Thinking ahead
  - Refactoring

Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*.
- We aim for *loose coupling*.
- Loose coupling makes it possible to:
  - understand one class without reading others;
  - change one class without affecting others.

Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has *high cohesion*.
- We aim for high cohesion.
  - A method should be responsible for one and only one well defined task.
  - Classes should represent one single, well defined entity.
- High cohesion makes it easier to:
  - understand what a class or method does,
  - use descriptive names,
  - reuse classes or methods.

Code duplication

- Code duplication
  - is an indicator of bad design,
  - makes maintenance harder,
  - can lead to introduction of errors during maintenance.

Responsibility-driven design

- Question: where should we add a new method (to which class)?
- Each class should be responsible for manipulating its own data.
- The class that owns the data should be responsible for processing it.
- RDD leads to low coupling.

Thinking ahead

- When designing a class, we try to think what changes are likely to be made in the future.
- We aim to make those changes easy.

Refactoring

- When classes are maintained, often code is added.
- Classes and methods tend to become longer.
- Every now and then, classes and methods should be *refactored* to maintain cohesion and low coupling.

Refactoring and testing

- When refactoring code, separate the refactoring from making other changes.
- First do the refactoring only, without changing the functionality.
- Test before and after refactoring to ensure that nothing was broken.

Summary

- Quality of code requires much more than just performing correct at one time.
- Code must be understandable and maintainable.
- Good quality code avoids duplication, displays high cohesion, low coupling.
- Coding style (commenting, naming, etc.) is also important.
- There is a big difference in the amount of work required to change poorly structured and well structured code.