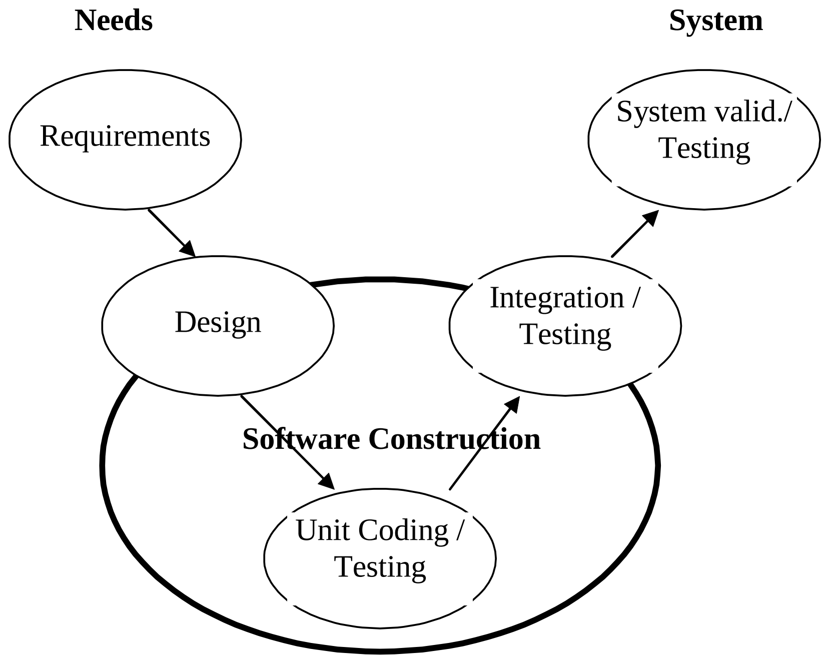
FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 1 – SUPPORT PRESENTATION (PART 1)

Outline

- Software development process
- Java Collections Framework
- Composite Design Pattern

Software Development Process

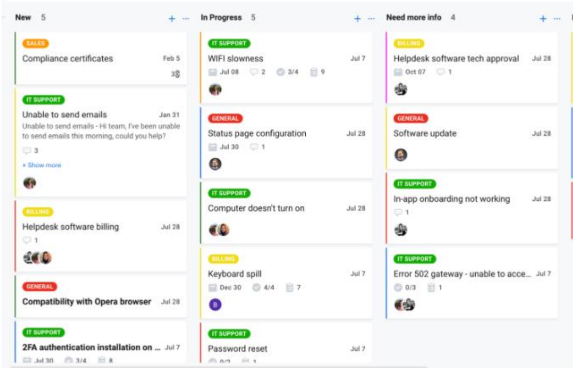


Problem and solution

PROBLEM: “Task management on paper is difficult and time consuming.”

How to design and implement the solution?

SOLUTION: Task Management Application

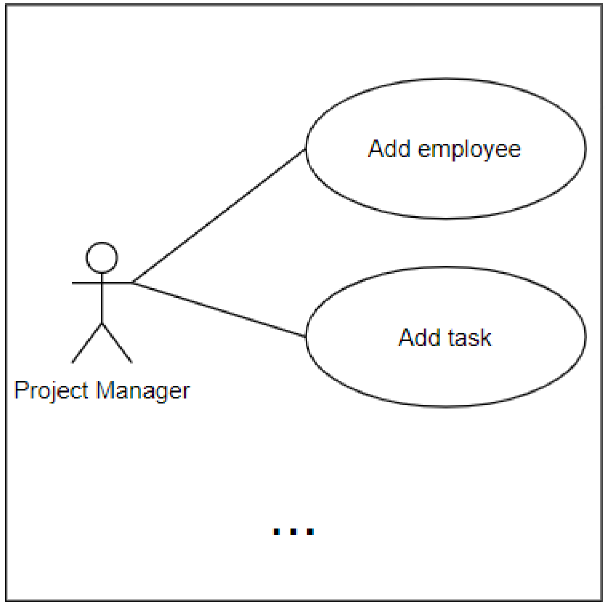


1. Clearly state the main objective and the sub-objectives required to reach it.
2. Analyze the problem and define the functional and non-functional requirements.
3. Design the solution
4. Implement the solution
5. Test the solution

Objectives

- Main objective
  - Design and implement a task management application with a dedicated graphical interface through which the project manager can manage employees and tasks.
- Sub-objectives
  - Analyze the problem and identify requirements
  - Design the task management application
  - Implement the task management application
  - Test the task management application

Analysis



Fragment of the Use Case Diagram

Define requirements

Functional requirements:

- The application should allow the project manager to add a new employee
- The application should allow the project manager to add a new task
- The application should allow the project manager to assign a task to an employee
- … what other functional requirements can you define? …

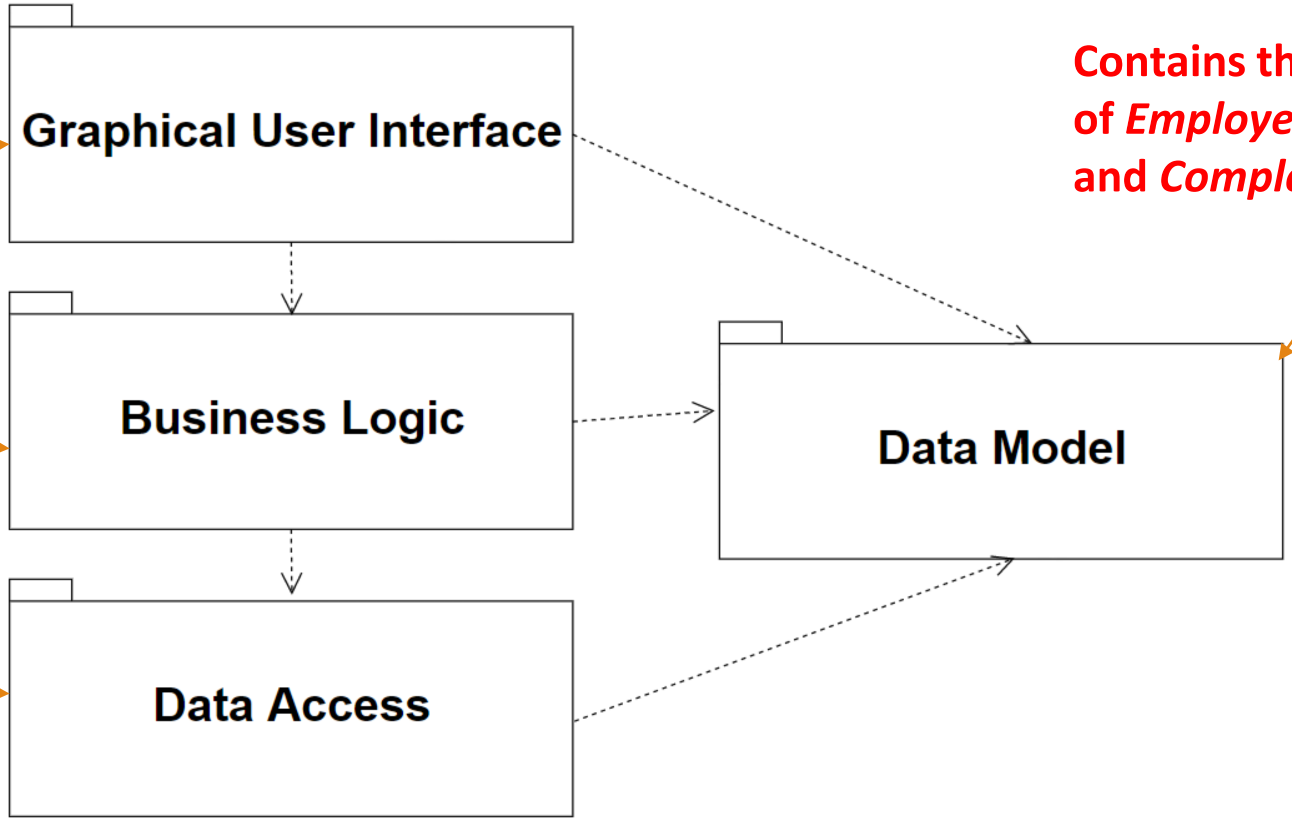
Non-Functional requirements:

- The task management application should be intuitive and easy to use by the user
- … what other non-functional requirements can you define? …

Detailed Design

Division into sub-systems/packages

Contains the implementation of the GUI



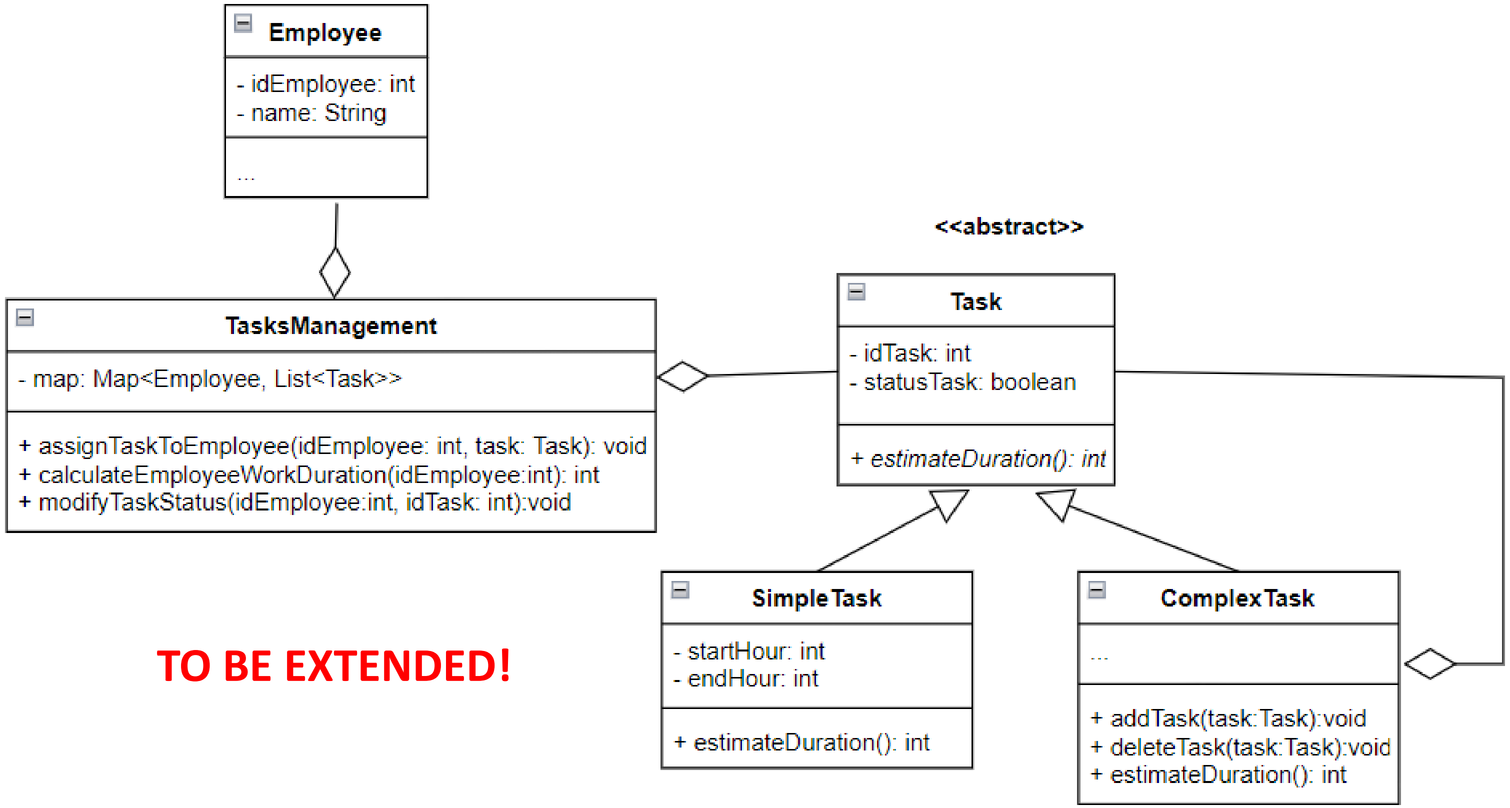
Contains the implementation of *Employee*, *Task*, *SimpleTask* and *ComplexTask*

Contains the implementation of *TaskManagement* and *Utility*

Contains the class for persisting data using serialization

Design

Division into classes and routines



TO BE EXTENDED!

! When defining the classes think about **ABSTRACTION, INHERITANCE,** and **ENCAPSULATION**

Java Collections Framework

Java Collections Framework

- Unified architecture for representing and manipulating collections
  - Collection = object that contains other objects (i.e., collection elements)
    - Collection elements can be added / removed / manipulated in the collection
- Benefits
  - Reduces programming effort; increases program speed and quality; allows interoperability among unrelated APIs; fosters software reuse

Collection types

Collection type	Description	Interface
Bag	Most general form of collections; it is unordered and allows duplicate elements	Collection
Set	Does not contain duplicate elements; can be sorted	Set
List	Ordered collection of indexed elements; allows duplicate elements	List
Map	Unordered collection of associations (key, value) – the key must be unique, the value can be any entity; can be sorted	Map

Implementation Data Structure Support

Backing data structure	Targeted collection
Array	ArrayList, many Queue / Deque and Hashtables implementations
Linked List	LinkedList, LinkedBlockingQueue, ConcurrentLinkedQueue HashSet and LinkedHashSet
Hash Table	HashSet, LinkedHashSet, HashMap, LinkedHashMap, WeakHashMap, IdentityHashMap, ConcurrentHashMap
Tree	TreeSet, TreeMap, PriorityQueue, PriorityBlockingQueue

Hash table as backing data structure

- **Hash Table**
  - Backing data structure for HashSet, **LinkedHashSet**, **HashMap**, HashTable, **LinkedHashMap**, etc.
  - Used to implement an associative array (by mapping keys to values) with constant access time to its elements
    - Constant access time => no repetitive structures => direct memory access
  - The keys will be used as indexes in an array: store the pair (key, value) as

    bucket[key]=value
  - The elements of the array are called **buckets**
  - **The problem with this approach is the large memory allocated and unused if the key set is sparse**
    => **Solution:** define a hash function to reduce the key set to a smaller set of size N

    hash : Keys -> {1..N}
    - **The pair (key, value) will be stored as:**

      bucket[hash(key)] = value

!

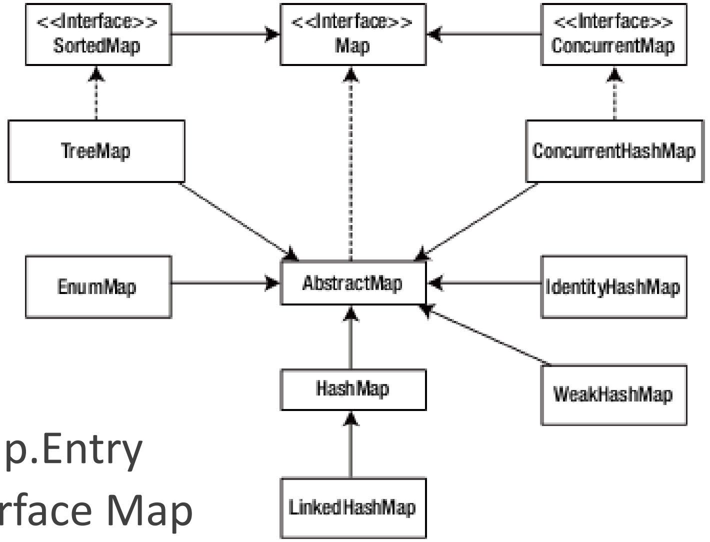
- **The hash function can lead to collisions when hash(key1) = hash(key2)**

Solved with

- **Open Addressing** : probe the next free space from the array in a given sequence
- **Chaining:** store a list in a bucket. Add all elements with the same hash value in the corresponding list

Java Map Interface

- **Java Map Interface**
  - Map
    - Object that maps keys to values
    - A (key, value) pair is an entry in the Map
    - No duplicate keys are allowed
    - One key maps to at most one value
  - Collection of Entries
    - An Entry is specified by the interface Map.Entry
      - Map.Entry - inner interface of the interface Map
  - Main Map implementations
    - Unsorted: HashMap, LinkedHashMap (inherits from HashMap)
    - Sorted: TreeMap – ordered by key
  - Iteration
    - Has no iterator method
    - **keySet()**, **entrySet()** methods return Set; **values()** method returns Collection -> Set and Collection can be iterated




```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    // provides Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Java HashMap

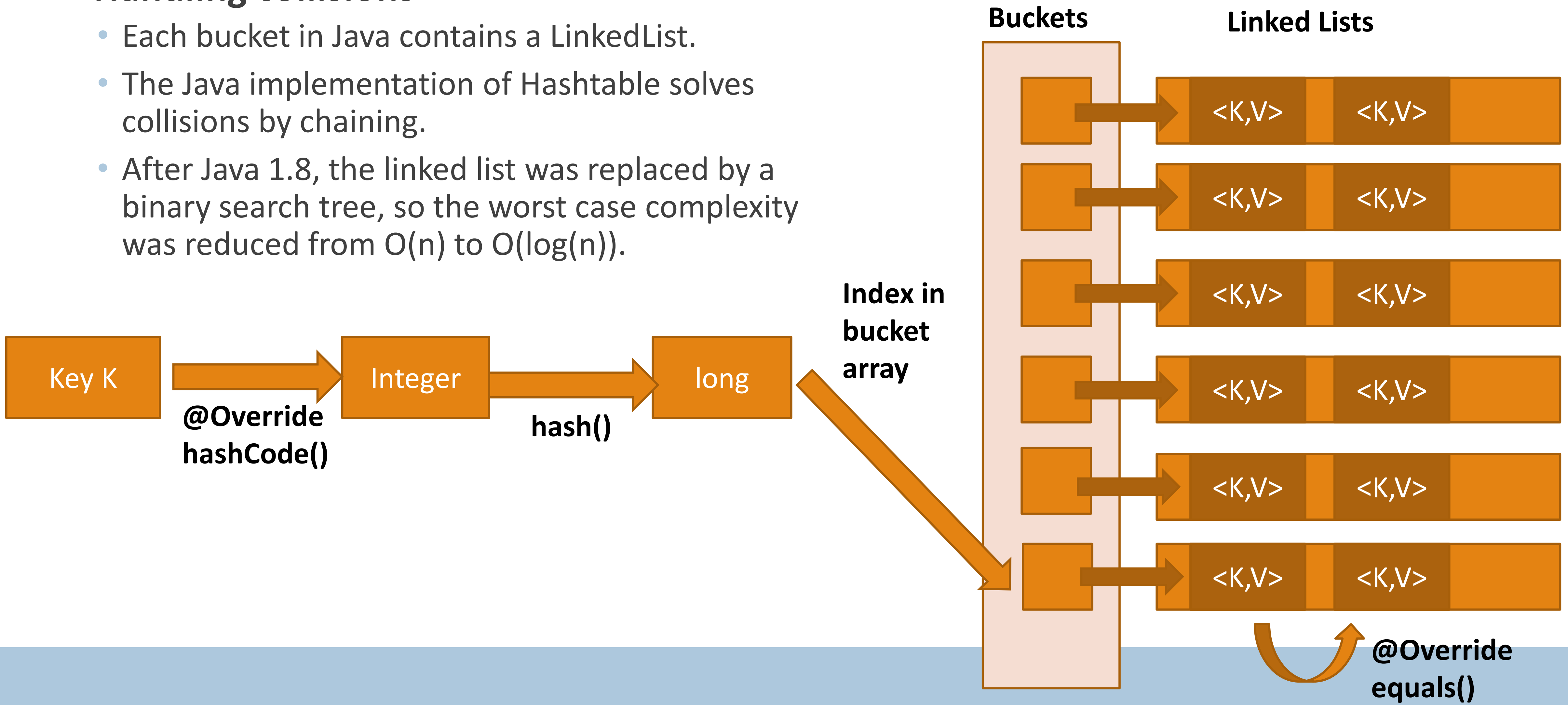
- Works on the principle of **hashing**
  - **Hashing** = assigning a unique code for any variable/object after applying any formula/algorithm on its properties
  - The **Hash function** should return the same hash code each and every time when the function is applied on same or equal objects => two equal objects must produce the same hash code
- Stores instances of the Entry class in an array:

```
transient Entry[] table;
```

```
static class Entry<K ,V> implements Map.Entry<K, V> {
  final K key;
  V value;
  Entry<K ,V> next;
  final int hash;
  ...//More code goes here
}
```

Java HashMap

- **Handling collisions**
  - Each bucket in Java contains a LinkedList.
  - The Java implementation of Hashtable solves collisions by chaining.
  - After Java 1.8, the linked list was replaced by a binary search tree, so the worst case complexity was reduced from O(n) to O(log(n)).



Java HashMap

- **Importance of equals() and hashCode()**

```
public V put(K key, V value) {
  if (key == null)
  return putForNullKey(value);
  int hash = hash(key.hashCode());
  int i = indexFor(hash, table.length);
  for (Entry<K , V> e = table[i]; e != null;
                               e = e.next){
    Object k;
    if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
      V oldValue = e.value;
      e.value = value;
      e.recordAccess(this);
      return oldValue;
    }
  }
  modCount++;
  addEntry(hash, key, value, i);
  return null;
}
```

Java Map Interface

- **Iteration examples**

```
Map<String, String> teacherToCoursesMap = new HashMap<String, String>();
teacherToCoursesMap.put("John Doe", "Distributed Systems");
teacherToCoursesMap.put("Mary Jones", "Mathematics");
teacherToCoursesMap.put("Ann Smith", "Physics");
```

Iterate over Map.entrySet() using the for-each loop

```
for(Map.Entry<String, String> entry: teacherToCoursesMap.entrySet()){
            System.out.println("Teacher=" + entry.getKey() + "; " +
                    "Course=" + entry.getValue());
        }
```

Iterate over keys or values using the for-each loop

```
for(String teacher: teacherToCoursesMap.keySet()){
     System.out.println("Teacher=" + teacher);
}
for(String course: teacherToCoursesMap.values()){
     System.out.println("Course=" + course);
}
```

Iterate over Map.Entry<K, V> using iterators

```
Iterator<Map.Entry<String, String>> iterator = teacherToCoursesMap.entrySet().iterator();
while(iterator.hasNext()){
    Map.Entry<String, String> entry = iterator.next();
    System.out.println("Teacher=" + entry.getKey() + " , Course=" + entry.getValue());
}
```

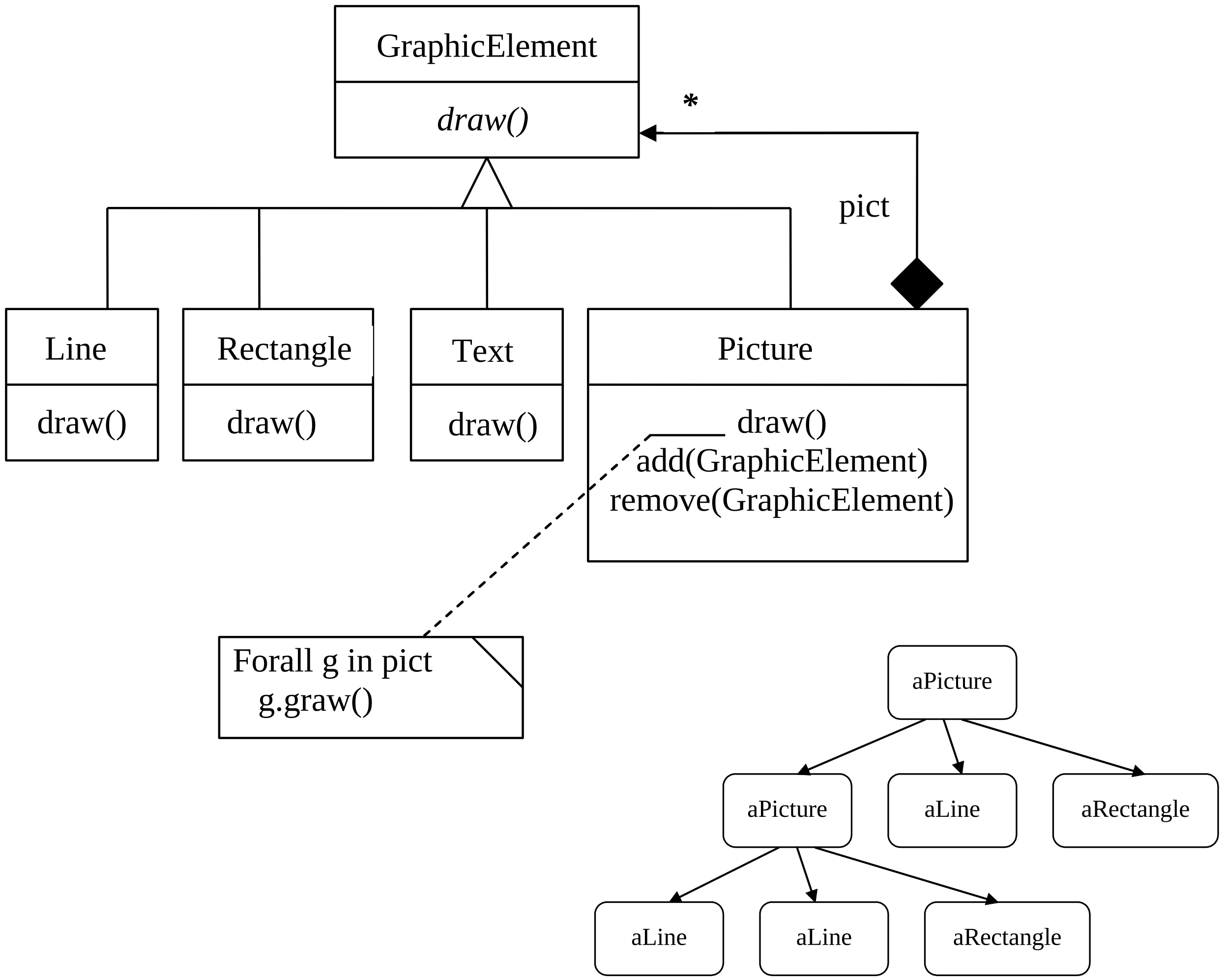
Map Data structures comparison

Property	HashMap	HashTable	LinkedHashMap	TreeMap
Synchronization or Thread Safe	No	Yes	No	No
Null keys and null values	One null key and any number of null values	No	One null key and any number of null values	Only values
Iterating the values	Iterator	Enumerator	Iterator	Iterator
Iterator type	Fail fast iterator	Fail safe iterator	Fail fast iterator	Fail fast iterator
Interfaces	Map	Dictionary	Map	Map, NavigableMap, SortedMap
Internal implementation	Hashtable with buckets	Hashtable with buckets	Hashtable with double-linked buckets	Red-Black Tree
Get/Put average Complexity	O(1)	O(1)	O(1)	O(log(n))
Get/Put worst complexity	O(n)	O(n)	O(n)	O(log(n))
Space Complexity	O(n)	O(n)	O(n)	O(n)
Order	No guarantee that order will remain constant over time	No guarantee that order will remain constant over time	Insertion-order	Sorted according to natural ordering of the keys

Composite Design Pattern

Composite Design Pattern

- Intention
  - Represents part-whole hierarchies as a result of object composition
  - Also known as recursive composition pattern
- Motivation
  - Atomic elements and containers
  - Treating atomic elements and containers
    - in the same way
    - in a different way
  - Composite pattern main feature
    - abstract class that represents both primitives and their containers



Composite Design Pattern

- Participants
  - Component
    - Declares the interface for the objects in Composition
  - Leaf
    - Represents leaf (atomic) objects in the composition
    - Defines the behavior for the primitive objects in the composition
  - Composite
    - Defines behavior for components having children
    - Stores child components
  - Client
    - Manipulates objects in the composition through the Component interface