Overview

- Sets
- equals( ) and hashCode( )
- Maps
- Java language features
  - Generic programming & generic classes
  - Inner classes
  - Clone
Set Data Structures

- No relationship between elements

Types of sets
- Set
- Map
- Hash Table

Sets

Properties
- Collection of elements without duplicates
- No ordering (i.e., no front or back)
- Order in which elements added doesn’t matter

Implementation goal
- Offer the ability to find / remove element quickly
- Without searching through all elements
# How Do Sets Work in Java?

- Finding matching element is based on `equals()`
- To build a collection for a class
  - Need to define your own `equals(Object)` method
  - Default `equals()` uses reference comparison
    - i.e., `a.equals(b) → a == b`
    - `a`, `b` equal only if reference to same object
  - Many classes have predefined `equals()` methods
    - `Integer.equals()` → compares value of integer
    - `String.equals()` → compares text of string

## Set Concrete Classes

- **HashSet**
  - Elements must implement `hashCode()` method
- **LinkedHashSet**
  - HashSet supporting ordering of elements
  - Elements can be retrieved in order of insertion
- **TreeSet**
  - Elements must be comparable
    - Implement `Comparable` or provide Comparator
  - Guarantees elements in set are sorted
Hashing in Java

**hashCode( ) method**
- Returns an int
- Used as efficient approximation of equals( )

**Object class has default hashCode() method**
- Usually just location of object in memory
- Usually need to override definition to work with new equals( )

**“Contract” between hashCode( ) and equals( )**

---

Java hashCode( ) Contract

**hashCode()**
- Must return same value for object in each execution, provided no information used in equals( ) comparisons on the object is modified

**equals()**
- if a.equals(b) == true, then must guarantee
  a.hashCode( ) == b.hashCode( )
- If a.hashCode( ) == b.hashCode( )
  - Does not imply a.equals(b)...
    ...though Java libraries are more efficient if true

**More on hashing later…**
Map Definition

- Map (associative array)
  - Unordered collection of keys
  - For each key, an associated object
  - Can use key to retrieve object
- Can view as array indexed by any (key) value
  - Example
    A["key1"] = ...

Map Interface Methods

- Methods
  - void put( Key, Object ) // inserts element
  - Object get( Key ) // returns element
  - void remove( Key ) // removes element
  - Boolean containsKey( Key ) // looks for key
  - Set keySet( ) // entire set of keys
Map Properties

- Map keys & map objects
  - Can also treat keys & values as collections
  - Access using keySet( ), values( )
  - Aliasing
    - Each key refers only a single object
    - But object may be referred to by multiple keys
  - Keys & values may be of complex type
    - Map<Object Type1, Any Object Type2>
    - Including other collections, maps, etc…

Map Implementation

- Implementation approaches
  - Two parallel arrays
    - Unsorted
    - Sorted
  - Linked list
  - Binary search tree
  - Hash table

- Java Collections Framework
  - TreeMap  → uses red-black (balanced) tree
  - HashMap  → uses hash table
Java Collections Map Hierarchy

SortedMap → AbstractMap → TreeMap

HashMap → LinkedHashMap

More Java Language Features

- Generic programming & generic classes
- Inner classes
- Cloning
Generic Programming

- **Generic programming**
  - Defining constructs that can be used with different data types
  - i.e., using same code for different data types

- **Implemented in Java through**
  1. Inheritance → A extends B
  2. Type variables → \(<A>\)

---

Generic Programming Examples

- **Inheritance**
  ```java
  Class A {
    doWork( A x ) { … }
  }
  Class B extends A { … }
  
  A w1 = new A( );
  B w2 = new B( );
  w1.doWork( w1 );
  w2.doWork( w2 );
  ```

- **Type Variables**
  ```java
  Class W<\(T\)> {
    doWork( T x ) { … }
  }
  
  Class A { … }
  Class B { … }
  
  W<A> x1 = new W<A>( );
  W<B> x2 = new W<B>( );
  A w1 = new A( );
  B w2 = new B( );
  x1.doWork( w1 );
  x2.doWork( w2 );
  ```
Generic Class

- Class with one or more type variables
  - Example → class ArrayList<E>

- To use generic class, provide an actual type
  - Valid types
    - Class → ArrayList<String>
    - Interface → ArrayList<Comparable>
  - Invalid types
    - Primitive type → ArrayList<int>
      (use wrappers) → ArrayList<Integer>

Defining a Generic Class

- Append type variable(s) to class name
  - Use angle brackets → ClassName<type variable>

- Can use any name for type variable
  - But typically single uppercase letter → T, E, etc…

- Use the type variable to define
  - Type of variables
  - Type of method parameters
  - Method return type
  - Object allocation
Example Generic Class

Example

```java
public class myGeneric<T> {
    private T value;
    public myGeneric( T v ) { value = v; }
    public T getVal() { return value; }
    public void setVal( T newV ) { value = newV; }
}
```

Inner Classes

Description

- Class defined in scope of another class

Property

- Can directly access all variables & methods of enclosing class (including private fields & methods)

Example

```java
public class OuterClass {
    private Object value;
    public class InnerClass {
        ...Object x = value;
    }
}
```
Java – Cloning

Cloning
- Creating an identical copy

Cloneable interface
- Supports clone( ) method
- Returns copy of object
  - Copies all of its fields
  - Does not clone its fields
  - Makes a shallow copy

Effect of clone( )
- Creates new object
  - X.clone( ) != X
- Same class
  - X.clone.getClass( ) == X.getClass( )
- Modification to X no longer affect X.clone( )
Java Clone Comparison

Example (X.f = Z)

X

X

Z

X

X

Z

Y = X

Y = X.clone()

X

X

Z

Z

X

X

Z

Z
Java Clone Comparison

Example (X.f = Z)
- Y = X; X.f = A
- Y = X.clone(); X.f = A

[Diagram showing the comparison of clone methods for different scenarios]