Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Recommended practices in OOP software development

T. Cioara, V. Chifu, C. Pop
2025

Interface Separation

When the class functionality can be implemented in different ways => separate the interface from the implementation

Advantages

- The implementation details are hidden
- The implementation can be changed easily

List definition: ordered collection whose elements are accessed through an index and whose size can grow as needed.

List alternatives: linked list, array list, stack, vector, ….

interface

Interface List

implementation

public class LinkedList implements List{…}

public class ArrayList implements List{…}

…

Immutable Classes and Objects

Immutable objects have their state set during the construction process and never changes afterwards.

How to enforce a class to generate immutable objects?

- Define its fields as final
- Have one (or more) class constructor(s) with parameters that assign values to all instance variables
- Remove mutator (setter) type methods

Examples of immutable classes from Java

- Wrapper classes (i.e., Integer, Double, Boolean, etc.), class String

```
public class Employee { // immutable class
    private final String employeeID;
    private final String firstName;
    private final String lastName;

     // constructor – assigns values to all fields
     public Employee(String id, String first, String last)
     {
         eployeeID = id;
         firstName = first;
         lastName = last;
     }

      public int getEmployeeID() { return
         Integer.parseInt(employeeID); }

      // remove set method
      public void setEmployeeID(int id) {
         employeeID = Integer.toString(id); }
    …
}
```

Immutable Classes and Objects

Advantages

- Thread-safety
- Easier to write, use and reason about code
- Better identify class invariants
- No conflicts among objects => easier to parallelize programs
- References to immutable objects can be cached as they are not going to change
- No invalid states – state of an immutable object always remains the same (advantage for high security)
- Better code testing => increasing robustness and error free code
- Increase readability and maintainability

Disadvantages

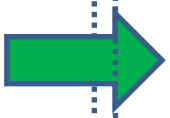
- Whenever you need to modify an object, you must create a new one
- **They require a separate object for each distinct value**

Immutable Classes and Objects

- **Having fields as final doesn't mean that the class is immutable**

- *final* only forbids us from changing the reference the variable holds,
- doesn't protect us from changing the internal state of the object it refers to by using its public API

```
public class Student {
   private final List<TestScore> testScores;
   private final String name;

   public Student(List<TestScore> scores,
                  String name) {
      this.testScores = scores;
      this.name = name;
   }

   public void addScore(TestScore sc) { … }

   public List<TestScore> getTestScores() {
         return testScores;
   }

 // … all setter methods are removed

}
```



Immutable Class

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Student {
   private final List<TestScore> testScores;
   private final String name;
   public Student(List<TestScore> scores, String name)
   {
    this.testScores = Collections.unmodifiableList (
              new ArrayList<TestScore>(scores));
    this.name = name;
   }

   public String getName() { return name;}

   public List<TestScore> getTestScores() {
        return testScores;
   }
   // …
}
```

Immutable Classes and Objects

- **Having fields as final doesn't mean that the class is immutable**

```
import java.util.*;
public class Employee {
 // instance variables
 private final String name;
 private final double salary;
 private final Date hireDate;
 public Employee (String aName, double sal,
                  Date aHireDate) {
    name = aName;
    salary = sal;
    hireDate = aHireDate;
 }
 public String getName() { return name;}

 public double getSalary(){return salary; }

 public Date getHireDate() {
    return hireDate; // Date is mutable
 }
}
```

```
import java.util.*;
public class TestEmployee {
 public static void main(final String[] args) {
   Date hd = new Date(); // current Calendar date

   Employee e = new Employee ("Ion", 1000, hd);
   Date ghd = e.getHireDate();

   System.out.println("Date before: " + ghd);

   ghd.setTime (1000999);  // changes ion's state

   System.out.println("Date after:"+ion.getHireDate());
 }
}
```

Solution

```
public Date getHireDate() {
  return (Date) hireDate.clone();
}
```

Favor immutable objects whenever possible!

Immutable Classes and Objects

- Java Records from JDK 15
  - Transparent carriers for immutable data
  - Construct that expresses a simple aggregation of values.
  - Focus on modeling immutable data rather than behavior.
  - Automatically implement data-driven methods such as equals and accessors.

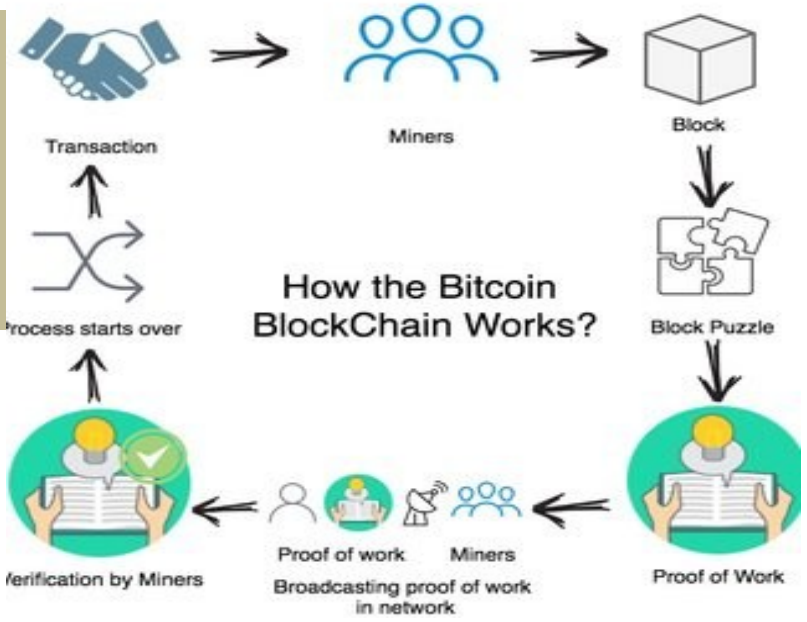
```
record Employee(String employeeID, String firstName, String lastName) { }
```

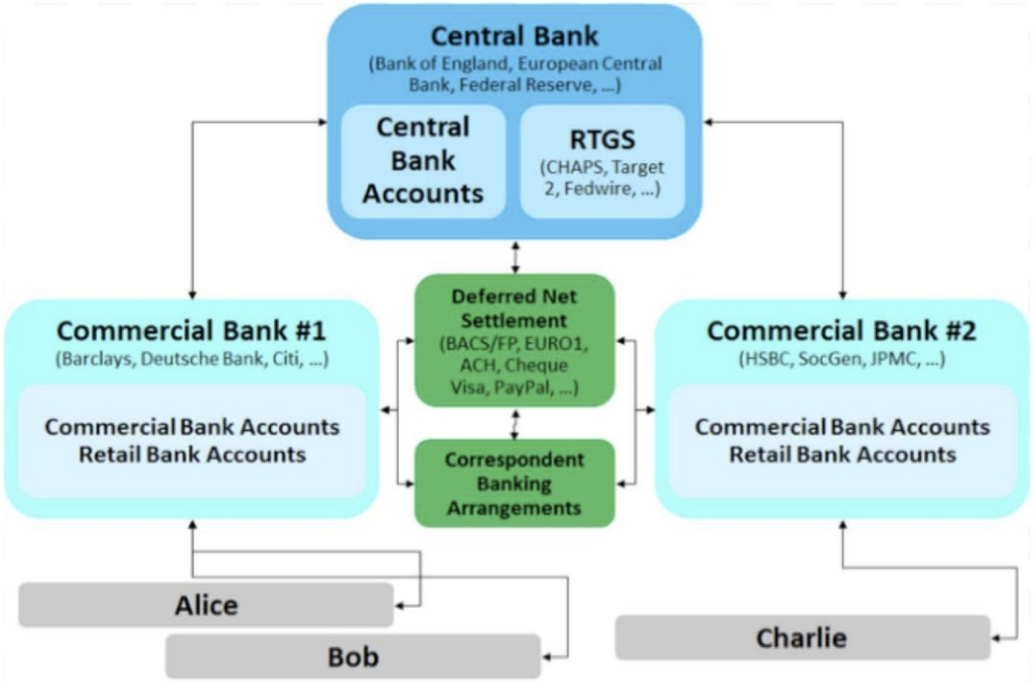
- Automatic generation of
  - public accessor methods with the same name and return type
  - private final fields with the same type
  - A *canonical constructor* whose signature is the same as the header
  - equals, hashCode and toString methods

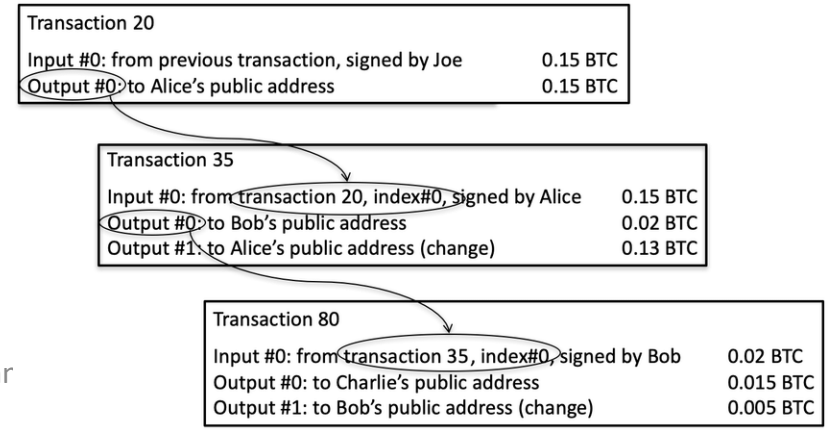
Immutable Classes and Objects

I've been working on a new electronic cash system that's fully **peer-to-peer**, with no trusted third party

Satoshi Nakamoto

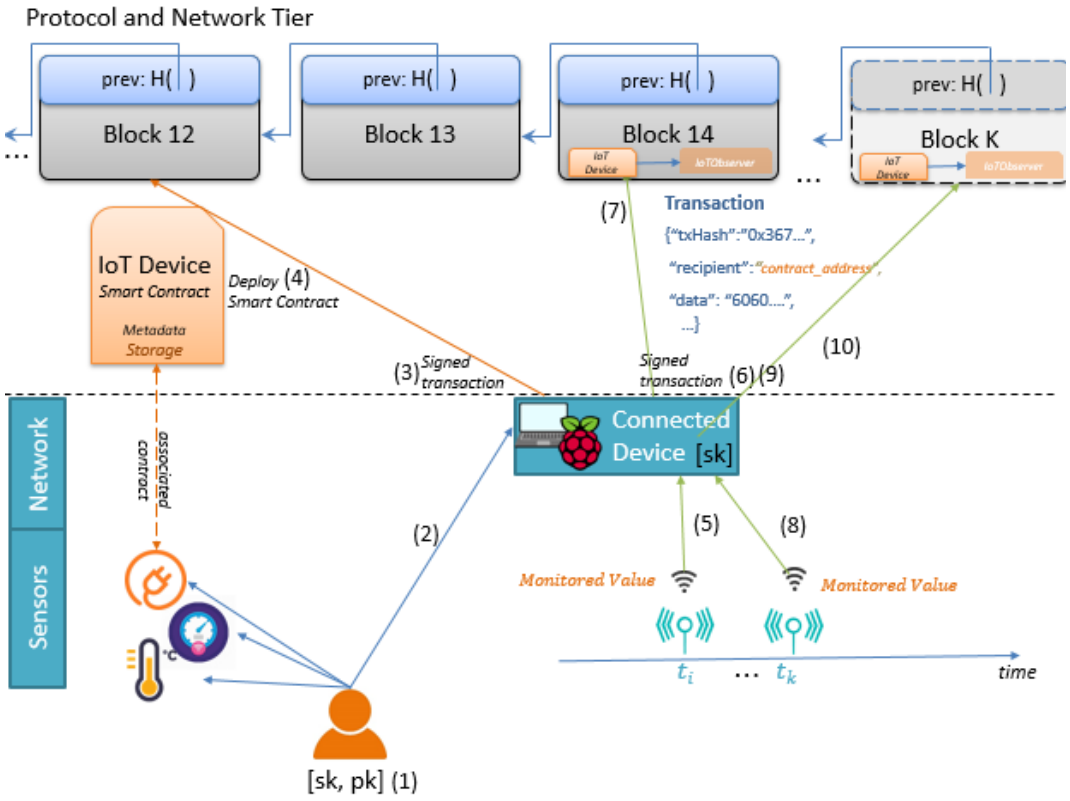


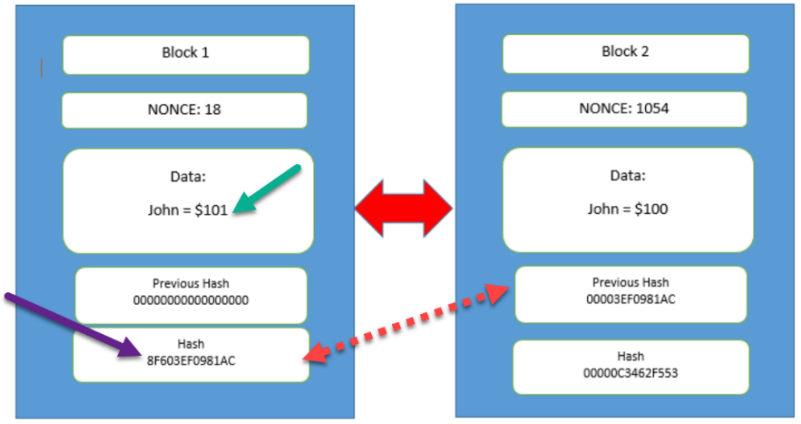




Immutable Classes and Objects

- Blockchain: Immutable, timestamped, hashed, distributed ledger





Quality features of classes / interfaces

Loose Coupling /
High Cohesion

Completeness

Convenience

Clarity

Consistency

Loose Coupling

Coupling = degree to which classes depend upon one another

- Tightly coupled - Two classes that are highly dependent
- Is inevitable - you cannot simply eliminate the interaction between classes
  - Classes must maintain references to one another and
  - Perform method calls

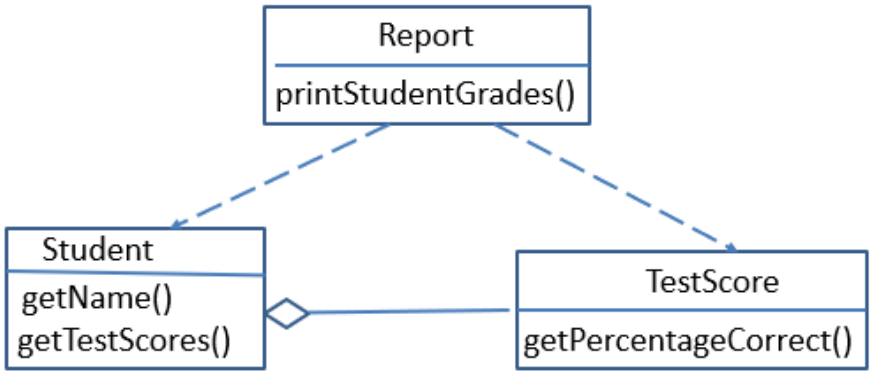
Solutions for low coupling

- Create a pure abstraction that handles the interaction between two classes or
- Shift the responsibility for the interaction to an existing class that you don't intend to make reusable

When implementing a class for reuse limit its dependencies on other classes as much as possible

Loose Coupling

- **Examples**



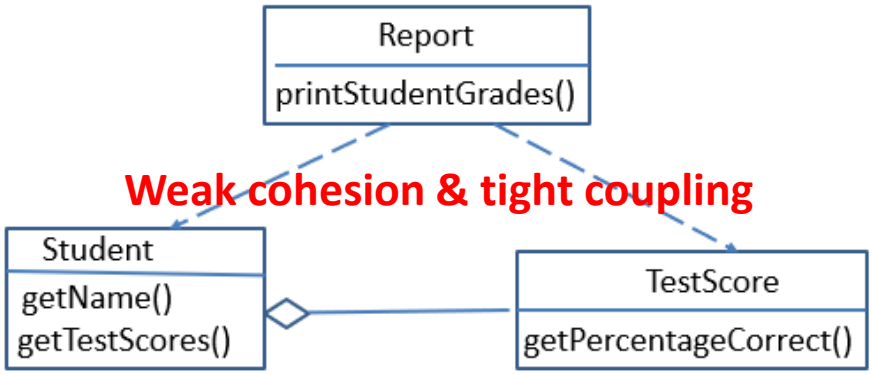

```
public class TestScore {
   private int percentCorrect;
   public TestScore(int percent) {
      this.percentCorrect = percent;
   }
   public int getPercentCorrect() {
      return percentCorrect;
   }
}
```

```
import java.util.List;
public class Student {
   private final List<TestScore> testScores;
   private final String name;
   public Student(List<TestScore> scores,
                          String name) {
      this.testScores = scores;
      this.name = name;
   }
   public String getName() { return name; }
   public List<TestScore> getTestScores() {
         return testScores;
   }
}
```

```
import java.util.List;
public class Report {
   public void printStudentGrades(Student[]  students)
{
      List<TestScore> testScores;
      int total;
      for (Student student : students) {
         testScores = student.getTestScores();
         total = 0;
         for (TestScore testScore : testScores) {
            total += testScore.getPercentCorrect();
         }
         System.out.println("Final grade for " +
            student.getName() + " is " +
            total / testScores.size());
      }}}
```

Loose Coupling

- **Examples – Problem**




```
public class TestScore {
   private int percentCorrect;
   public TestScore(int percent) {
      this.percentCorrect = percent;
   }
   public int getPercentCorrect() {
      return percentCorrect;
   }
}
```

```
import java.util.List;
public class Student {
   private final List<TestScore> testScores;
   private final String name;
   public Student(List<TestScore> scores,
                  String name) {
      this.testScores = scores;
      this.name = name;
   }
   public String getName() { return name; }
   public List<TestScore> getTestScores() {
         return testScores;
   }
}
```

```
import java.util.List;
public class Report {
   public void printStudentGrades(Student[]  students)
{
      List<TestScore> testScores;
      int total;
      for (Student student : students) {
         testScores = student.getTestScores();
         total = 0;
         for (TestScore testScore : testScores) {
            total += testScore.getPercentCorrect();
         }
         System.out.println("Final grade for " +
            student.getName() + " is " +
            total / testScores.size());
      }}}
```

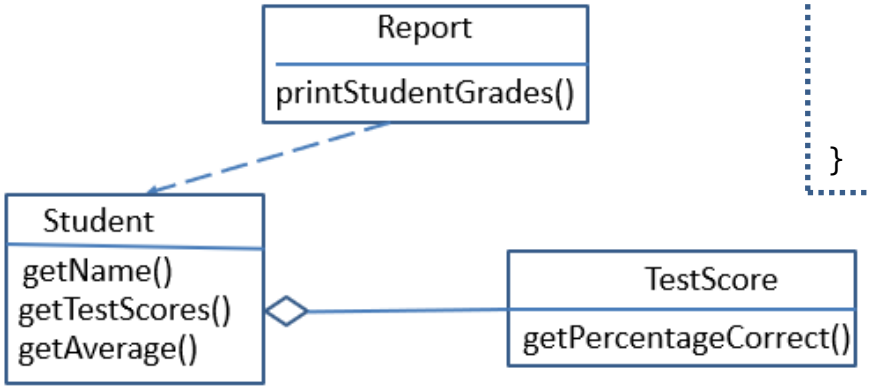
Loose Coupling

Problems & Solution

- Report - is coupled both to Student and to TestScore
- Unnecessary tight coupling
- Report - weak cohesion due to performing two functions:
  - printing a report and
  - calculating each student's average.

```
import java.util.List;
public class Student {
   // … previous code
   public int getAverage() {
      int total = 0;
      for (TestScore testScore : testScores) {
         total += testScore.getPercentCorrect();
      }
      return total / testScores.size();
   }}
```

```
public class Report {
   public void printStudentGrades (Student[] students) {
      for (Student student : students) {
        System.out.println("Final grade for " +
                  student.getName() +
                  " is " + student.getAverage());
      }
   }
}
```



High cohesion

Classes and interfaces

An interface isn't cohesive if

- Some set of closely related functions is split between interfaces
- Too much functionality to a class

Good rule of thumb

Responsibilities of a class limited enough that they can be outlined with a brief description

Condition for a class to be cohesive

Fields should be related to a single abstraction

Other Quality Features

- **Convenience**
  - The interfaces should provide convenient ways to accomplish common tasks
  - Example - common task of reading input from System.in
    - Before Java 5.0: "System in" must be wrapped into an InputStreamReader and then into a BufferedReader (**inconvenient**)
    - After Java 5.0: Scanner class solved this problem in a **more convenient** way
- **Consistency**
  - The operations should be consistent with respect to names, parameters and return values, and behavior

Constructor of GegorianCalendar in java.util:
`GregorianCalendar(int year, int month, int dayOfMonth)`
where:
- month: 0 ..11
- dayOfMonth: 1 .. 31

} **inconsistency**

Class Date
Defines the method **setTime()** for setting the Date (instead of a **setDate()** as it would be normal)
=> inconsistency

Other Quality Features

- **Clarity**
  - The interface of a class should not generate confusion
  - Example – Interface **ListIterator**

```
// Adding is intuitive

ListIterator<String> iterator = list.listIterator( ); // I ABC
iterator.next(); // A I BC
iterator.add("X" ); // AX I BC
```

```
// Remove is not intuitive as add
iterator.remove(); // A I BC

// Removes from the list the last element that was returned by next or previous.
```

Other Quality Features

- **Clarity and expressiveness**
  - Explaining intent in code instead of writing comments

```
//check if the client is eligible for full discount
if(client.Type == 2 && client.SubscribedToNewsletter) {…}
```

```
if(client.IsElligibleForFullDiscount()) {…}
```

- **Completeness**
  - Support all operations that are a part of the abstraction that the class represents.

Object Equality

Identity Equality

Implemented by default in class Object
o1.equals(o2)

```
public boolean equals(Object obj)
{
    return this==obj;
}
```

State-based Equality

Classes should override the **equals** method to implement a content-based equality

```
String s1 = new String ("Hi")
String s2 = new String ("Hi")
```

```
String s1= "Hi"
String s2 = "Hi"
```

```
s1==s2 vs s1.equals(s2)
```

```
public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        return (anObject instanceof String aString)
          && (!COMPACT_STRINGS || this.coder == aString.coder)
          && StringLatin1.equals(value, aString.value);
    }
```

Object Equality

- **Method equals contract for non-null object references**

Reflexivity	x.equals(x) should return true
Symmetry	x.equals(y) should return true if & only if y.equals(x) is true
Transitivity	if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
Consistency	multiple invocations of x.equals(y) consistently return true or consistently return false
Non-nulity	x.equals(null) should return false

Whenever the equals method is overridden, the hashCode method must also be overridden (equal objects must have equal hash codes).

Object Equality

- **State-based Equals - skeleton**

```
1   public class C {
2    // ... class resources
3
4      public boolean equals (Object o) {
5         if (o == this) return true;
6         if (!(o instanceof C)) return false;
7         C cObj = (C) o;
8         return ...; // logical test of equality
9      }
10  }
```

Step 1: Use == operator to check if the argument is a reference to this object.

Step 2: Use instanceof operator to check if the argument is of the correct type.

Step 3: Cast argument to the correct type.

Step 4: For each “significant” field in the class check to see if that field of the argument matches the corresponding field of this object.

Step 5: Check if the overridden equals method is symmetric, transitive and consistent.

Object Equality

- **State-based Equals – skeleton (step 4)**
  - For primitive fields p if(p != o.p) return false;
  - For object reference fields invoke equals recursively;
  - Some instance variables could contain null values
    - Avoid throwing NullPointerExceptions

      (field == null ? o.field == null : field.equals(o.field))
  - Exclude (not comparing) the following fields
    - temporarily fields,
    - derived fields (from other fields) or
    - nonessential fields

Object Equality

- **Example of method override**

```
public interface List {
   // mutators
   public void addElement(Object le, int i);
   public void addFirst(Object le);
   public void addLast(Object le);
   public Object remove(int i);
   public Object removeFirst();
   public Object removeLast();
   // getters (accessors)
   public Object getFirst();
   public Object getLast();
   public Object getElement(int i);
   public int getSize();
   // test
   public boolean isEmpty();
  // overrides
  public boolean equals (Object o);
}
```

If an interface does not extend another interface, the interface will implicitly declare a public abstract method for each public instance method from class Object.

Overriding the equals method

```
// LList represents a linked list implementation
public class LList implements List {
   …
    public boolean equals(Object o) {
      if (this == o) return true;
      if (o instanceof LList) {
         LList oList = (LList) o;
         if(this.getSize() == oList.getSize()) {
             for(int i = 0; i < this.getSize(); i++) {
                Object thisItem = this.getElement(i);
                Object oItem = oList.getElement(i);
                if(thisItem == null) {
                    if(oItem != null) { return false; }
                } else {
                    if(!thisItem.equals(oItem)) {
                  return false;
             }
               } //if
            } // for
            return true;
         } // if
        } // if
        return false;
    }
}
```

Object Equality

- **Not override equals**
  - Unique class instances – singleton classes
  - Doesn't matter whether the class provides a “logical equality”
  - A superclass has already overridden equals
    - Behavior inherited from the superclass is appropriate for this class.
  - Overriding only equals() method without overriding hashCode()
  - The class is private or package-private
    - Only when you are certain that its equals method will not be invoked
  - Just in case protection

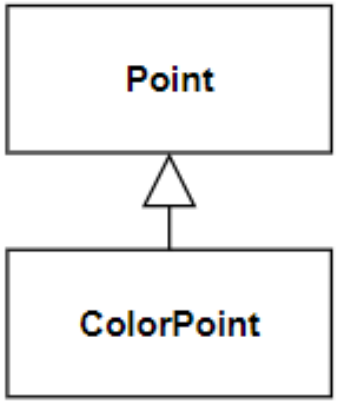
```
public boolean equals(Object o) {
  throw new UnsupportedOperationException();
}
```

Object Equality

- **Overriding the equals method in the context of inheritance**

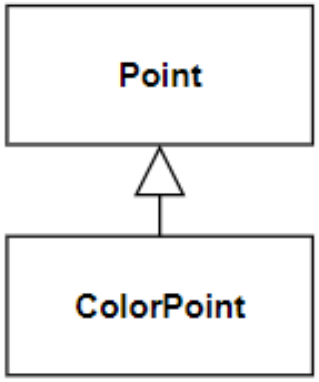
```
public class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x; this.y = y;
  }
  public boolean equals(Object o) {
    if (!(o instanceof Point) return false;
    Point p = (Point)o;
    return p.x == x && p.y == y;
  }
  // … rest of class resources
}
```

```
public class ColorPoint extends Point {
  private Color color;

  public ColorPoint(int x, int y, Color
                                    color) {
     super(x, y);
     this.color = color;
  }
  // … rest of class resources
  // method equals on following slides
}
```



Object Equality

- **Overriding the equals method in the context of inheritance**




```
public class ColorPoint extends Point {
  private Color color;
  public ColorPoint(int x, int y, Color color) {
     super(x, y);
     this.color = color;
  }
  // … rest of class resources

  public boolean equals(Object o) { //try 1
    if (!(o instanceof ColorPoint) return false;
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
  }
}
```

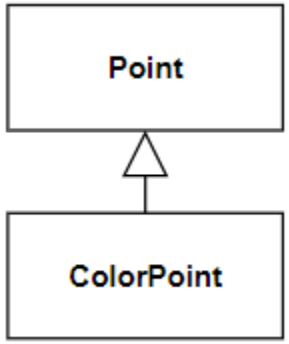
```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

p.equals(cp) => true
cp.equals(p) => false

Violates symmetry!

Object Equality

- **Overriding the equals method in the context of inheritance**




```
public class ColorPoint extends Point {
  private Color color;
  public ColorPoint(int x, int y, Color color) {
     super(x, y);
     this.color = color;
  }
  // … rest of class resources

  public boolean equals(Object o) { // try 2
    if (!(o instanceof Point)) return false;
    // If o is a normal Point, do a color-less comparison
    if (!(o instanceof ColorPoint)) return o.equals(this);
    // o is a ColorPoint; do a full comparison
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
  }
}
```

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

p1.equals(p2) and p2.equals(p3) => true

p1.equals(p3) => false

Transitivity violation

Object Equality

- **Overriding the equals method in the context of inheritance**
  - There is no simple way to preserve equals when extending an instantiable class and adding an extra attribute!
  - Solution: Favor composition over inheritance!

Even in java

java.sql.Timestamp subclasses java.util.Date (see Link)

- added nanoseconds field - equals violate symmetry
- problems if Timestamp and Date objects are used in the same collection

```
public class ColorPoint {
  private Point point;
  private Color color;

  public ColorPoint(int x, int y, Color color) {
    point = new Point(x, y);
    this.color = color;
  }

  public Point getPoint() { return point; }

  public boolean equals(Object o) {
    if (!(o instanceof ColorPoint)) return false;
    ColorPoint cp = (ColorPoint)o;
    return cp.getPoint().equals(point) &&

cp.color.equals(color);
  }

  // … The rest of class resources
}
```

Object Equality

- **Final recommendations**

R1	Override hashCode when overriding equals.
R2	Avoid equals method using unreliable resources.
R3	Don’t substitute Object type in equals method declaration.

Example: java.net.URL's equals method
- Relies on the IP addresses of the hosts in URLs being compared
- Translating a host name to an IP address can require network access, and it isn't guaranteed to yield the same results over time
  - Violate the equals contract, and it has caused problems in practice

public boolean equals(NotObjectClass o)
{ …}
- **Is not overriding Object equals**
- **Strong typed equals**

Object Hash Code

- hashCode method
  - Returns a hash code value for the object of invocation
  - Used by the hash-based collections

R1	When hashCode is invoked on the same object => same integer result
R2	hashCode invocation on two equal objects (equals method) => same integer result

Case of two unequal objects and hashCode

- if !o1.equals(o2) => hashCode(o1) might be equal to hashCode(o2)

Good hashtable performances when:

!o1.equals(o2) => hashCode(o1) != hashCode(o2)

Object Hash Code

- **Method skeleton**

```
1 public class C {
2     …
4     public int hashCode() {
5         int hash = 0; // cumulative
6         // for each field …
7         // … compute and combine the hash code
8         return hash;
9     }
10 }
```

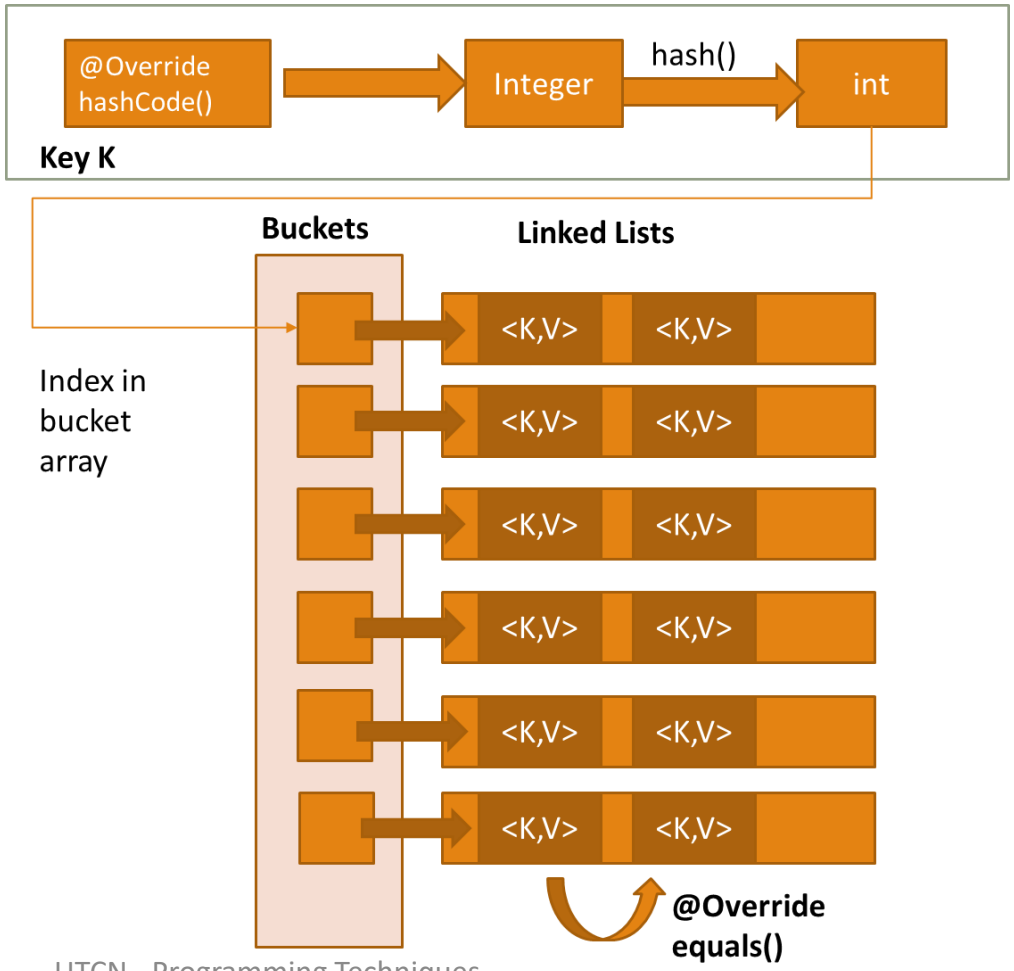
- Equal objects must have equal hash codes
- hashCode() should be overridden for all classes that overrides equals!
- Main rule violation of the general contract prevents proper operation with hash-based collections

Object Hash Code

- **How to compute the hash code of an object?**
  - Calculate a hash code for each significant fields
    - Primitive type field => convert to integer
    - Reference type field => call hashCode for that field
  - Combine the hash codes of all significant fields
    - Bitwise or: hash = hash << n | c, n is an arbitrary integer constant
    - Addition: hash = hash * p + c, p is a prime number (31)

Using equals and hashcode in Java HashMap

- **Java HashMap**




```
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;
}
```

Object Cloning

- Method clone
  - Default implementation: field by field copy or shallow copy
  - A class can have objects cloned only if it implements Cloneable
- Contract

R1	For any object x, the following expression is true: **x.clone() != x**
R2	For any object x, the following expression is true: **x.clone().getClass() == x.getClass()**
R3	For any object x, the following expression is true: **x.clone().equals(x)**

Method Skeleton

```
1   public class C implements Cloneable{
3    …
4    public Object clone() throws CloneNotSupportedException {
5       C clone = (C)super.clone();
6       // … do cloning of reference
7       // type fields for deep copy
8       return clone;
9    }
10  }
```

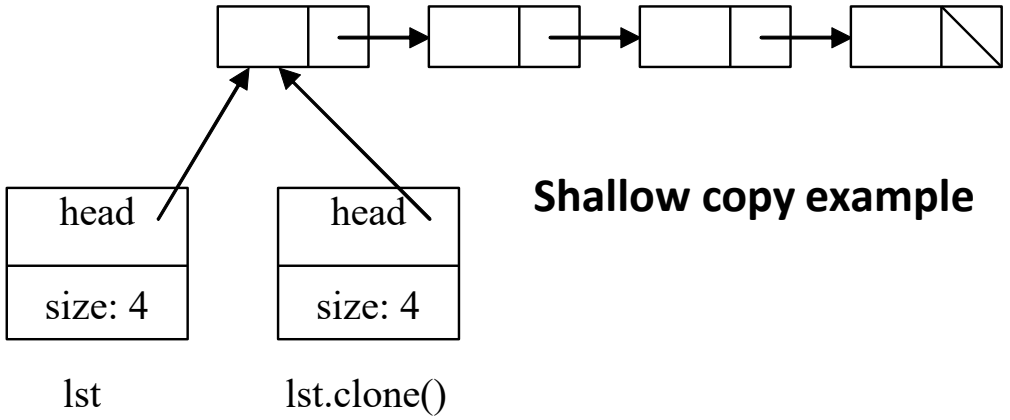
Overriding the **clone** method

Object Cloning

- **Shallow copy versus deep copy**

Shallow copy

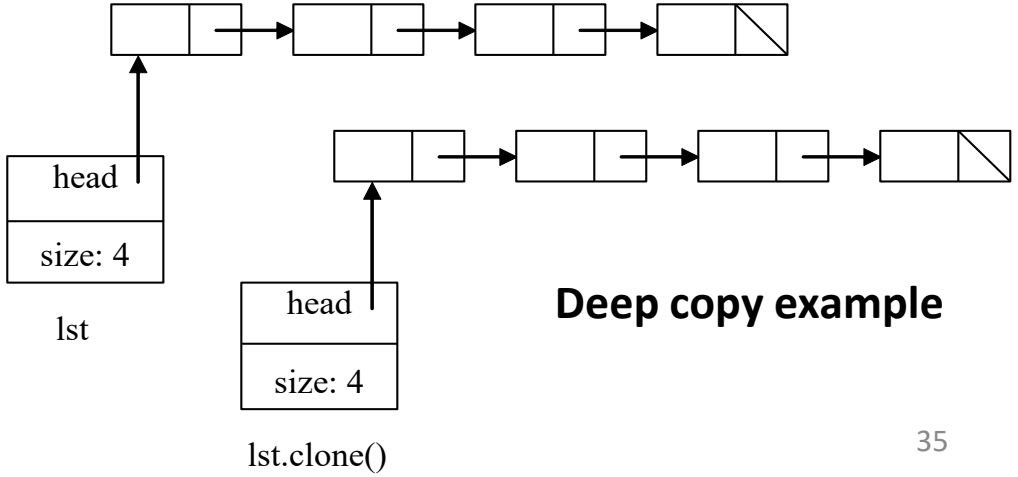
- Field by field copy
- Ok for primitive type fields
- For reference types - the referenced objects are not cloned except for objects that contain references to immutable objects and/or primitives



Shallow copy example

Deep copy

- Reference types are also cloned
- Useful for more complex object structures
- Your responsibility to implement the functionality



Deep copy example

Object String Representation

- toString method
  - Useful in testing and debugging
  - The result should include all object fields
  - Explicitly (in debug for example)
  - Implicitly whenever an object reference is specified as part of a string expression

The default implementation in Object

- displays the name of the object's class
- object's hash code value, separated by the at (@)
- Example C@28ccdaf5