Principles of Software Construction: Objects, Design and Concurrency

Java Collections (continued) and GUI Intro

15-214

toad

Spring 2013

Christian Kästner **Charlie Garrod**

School of Computer Science
institute for SOFTWARE RESEARCH

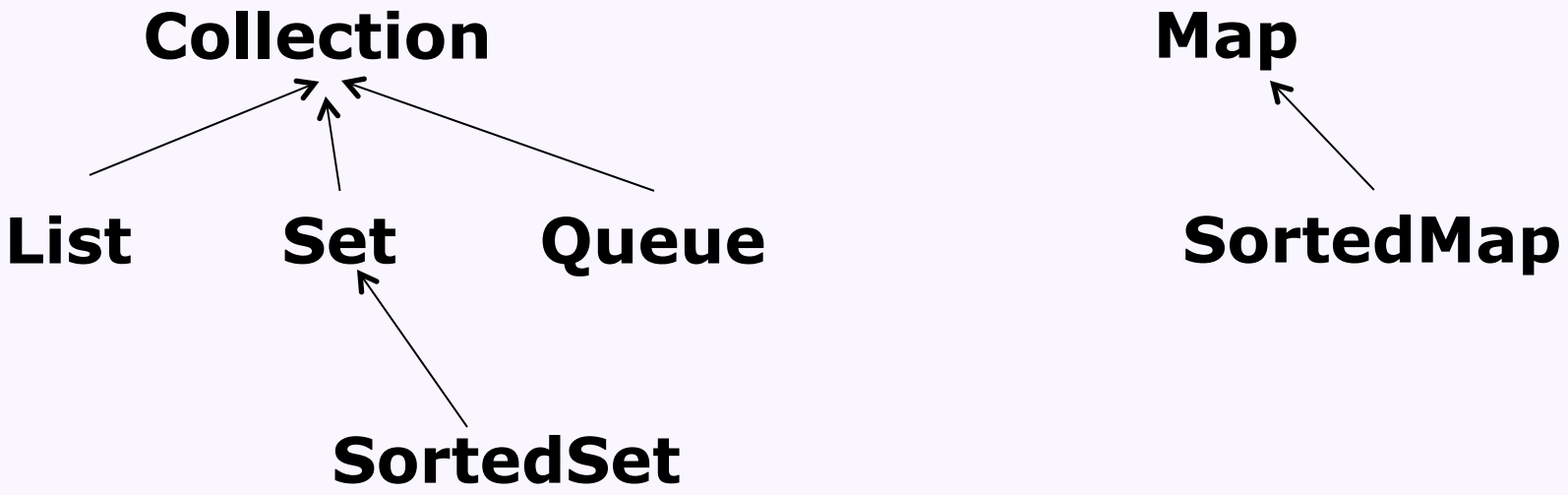
Administrivia

- Midterm in class next Tuesday
  - Review session Sunday 1 – 3 p.m., PH 100
  - Sample midterm and solutions coming soon
- Homework 3 grading expected by Saturday night
- Homework 4a due tonight
  - Feedback on your design by next Thursday…
  - …but you should start Homework 4b without waiting for feedback

Key topics from Tuesday

The Java Collections Framework

- Interfaces (in java.util)



- Default Implementations
  - ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue, HashMap, TreeMap, LinkedHashSet, LinkedHashMap, …
- Algorithms
  - min, max, sort, reverse, binarySearch, shuffle, rotate, …

The java.util.Collection<E> interface

```
boolean      add(E e);
boolean      addAll(Collection<E> c);
boolean      remove(E e);
boolean      removeAll(Collection<E> c);
boolean      retainAll(Collection<E> c);
boolean      contains(E e);
boolean      containsAll(Collection<E> c);
void         clear();
int          size();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
E[]          toArray(E[] a);
```

Today

- More Collections details
  - Sorting, …
- [A break]
- A brief introduction to GUIs

Sorting a Collection

- Use the Collections.sort method:

```
public static void main(String[] args) {
  List<String> lst = Arrays.asList(args);
  Collections.sort(lst);
  for (String s : lst) {
    System.out.println(s);
  }
}
```

- Abuse the SortedSet:

```
public static void main(String[] args) {
  SortedSet<String> set =
        new TreeSet<String>(Arrays.asList(args));
  for (String s : lst) {
    System.out.println(s);
  }
}
```

Sorting your own types of objects

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

- General contracts:
  - `a.compareTo(b)` should return:
    - `<0` if `a` is less than `b`
    - `0` if `a` and `b` are equal
    - `>0` if `a` is greater than `b`
  - Should define a total order
    - If `a.compareTo(b)` < 0 and `b.compareTo(c)` < 0, then `a.compareTo(c)` should be < 0
    - If `a.compareTo(b)` < 0, then `b.compareTo(a)` should be > 0
  - Should usually be consistent with `.equals`
    - `a.compareTo(b)` == 0 iff `a.equals(b)`

Comparable objects – an example

```
public class Integer implements Comparable<Integer> {
  private int val;
  public Integer(int val) { this.val = val; }
  …
  public int compareTo(Integer o) {
    if (val < o.val) return -1;
    if (val == o.val) return 0;
    return 1;
  }
}
```

- Aside: Why did I not just `return val – o.val`?

Comparable objects – another example

- Make Name comparable:

```
public class Name                                          {
  private String first;
  private String last;
  public Name(String first, String last) {   // should
    this.first = first;  this.last = last;   // check
  }                                          // for null
  …




}
```

- Hint: Strings implement `Comparable<String>`

Comparable objects – another example

- Make Name comparable:

```
public class Name implements Comparable<Name> {
  private String first;
  private String last;
  public Name(String first, String last) {   // should
    this.first = first;  this.last = last;   // check
  }                                          // for null
  …
  public int compareTo(Name o) {
    int lastComparison = last.compareTo(o.last);
    if (lastComparison != 0) return lastComparison;
    return first.compareTo(o.first);
  }
}
```

Alternative comparisons

```
public class Employee implements Comparable<Employee> {
   private Name name;
   private int  salary;
   …
}
```

- What if we want to sort Employees by name, usually, but sometimes sort by salary?

Alternative comparisons

```
public class Employee implements Comparable<Employee> {
  private Name name;
  private int  salary;
  …
}
```

- What if we want to sort Employees by name, usually, but sometimes sort by salary?
- Answer: There's an app^H^H^Hinterface for that

```
public interface Comparator<T> {
  public int     compare(T o1, T o2);
  public boolean equals(Object obj);
}
```

Writing a Comparator object

```
public class Employee implements Comparable<Employee> {
  private Name name;
  private int  salary;
  public int compareTo(Employee o) {
    return name.compareTo(o.name);
  }
}

public class EmpSalComp implements Comparator<Employee> {
  public int compare (Employee o1, Employee o2) {
    return o1.salary – o2.salary;  // Why is this OK?
  }
  public boolean equals(Object obj) {
    return obj instanceof EmpSalComp;
  }
}
```

Using a Comparator

- Order-dependent classes and methods take a Comparator as an argument

```
public class Main {
  public static void main(String[] args) {
    SortedSet<Employee> empByName =   // sorted by name
          new TreeSet<Employee>();

    SortedSet<Employee> empBySal =  // sorted by salary
          new TreeSet<Employee>(new EmpSalComp());
  }
}
```

Aside: The java.util.SortedSet<E> interface

- Extends java.util.Set<E>:

```
Comparator<E> comparator();
E             first();
E             last();
SortedSet<E>  subSet(E fromElement, E toElement);
SortedSet<E>  headSet(E toElement);
SortedSet<E>  tailSet(E fromElement);
```

- The `comparator` method returns null if the natural ordering is being used

The java.util.Collections class

- Standard implementations of common algorithms
  - binarySearch, copy, fill, frequency, indexOfSubList, min, max, nCopies, replaceAll, reverse, rotate, shuffle, sort, swap, …

```
public class Main() {
  public static void main(String[] args) {
    List<String> lst = Arrays.asList(args);
    Collections.sort(lst);
    for (String s : lst) {
      System.out.println(s);
    }
  }
}
```

The java.util.Collections class

- Standard implementations of common algorithms
  - binarySearch, copy, fill, frequency, indexOfSubList, min, max, nCopies, replaceAll, reverse, rotate, shuffle, sort, swap, …

```
public class Main() {
  public static void main(String[] args) {
    List<String> lst = Arrays.asList(args);
    int x = Collections.frequency(lst, "Charlie");
    System.out.println("There are " + x +
                       " students named Charlie");
  }
}
```

The java.util.Collections class

- Standard implementations of common algorithms
- An actual method declaration

```
static int binarySearch(
        List<? extends Comparable<? super T>> list,
        T                                     key);
```

An object of some type T to search for

A List of objects of some type that has a compareTo method that can take an object of type T as an argument

Today

- More Collections details
  - Sorting, …
- [A break]
- A brief introduction to GUIs

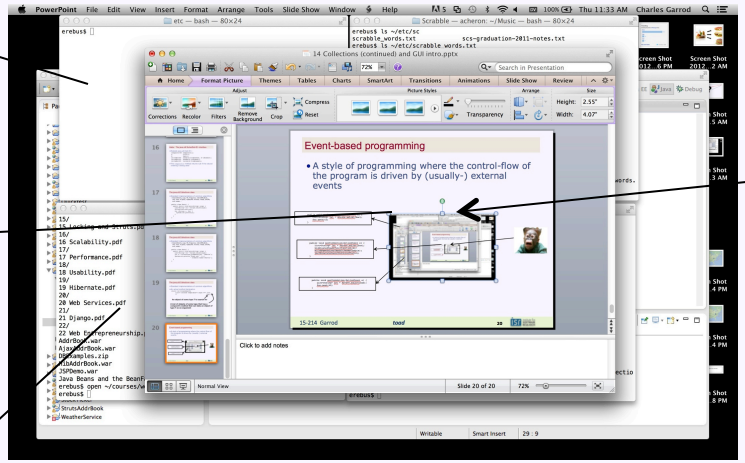
Event-based programming

- A style of programming where the control-flow of the program is driven by (usually-) external events

```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(42)
}
```

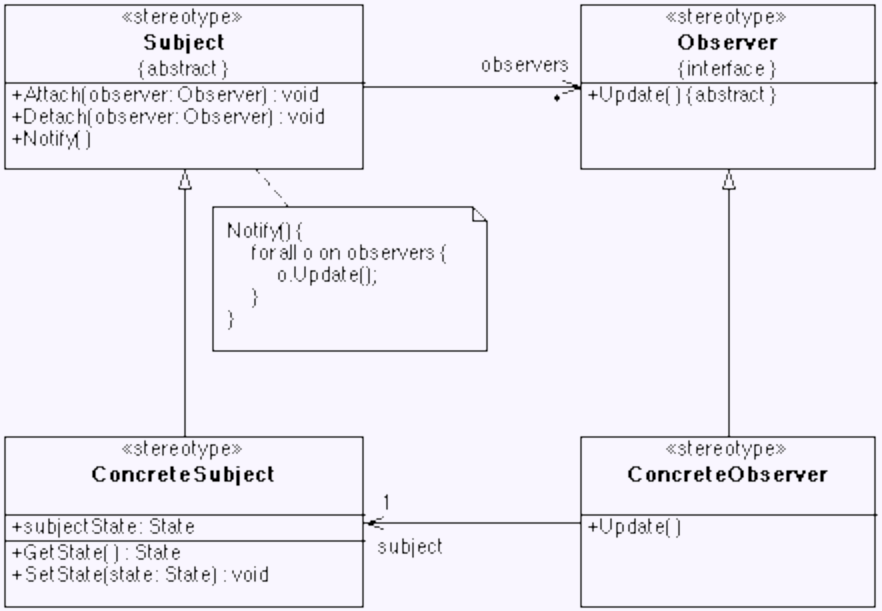
```
public void performAction(ActionEvent e) {
    bigBloatedPowerPointFunction(e);
    withANameSoLongIMadeItTwoMethods(e);
    yesIKnowJavaDoesntWorkLikeThat(e);
}
```

```
public void performAction(ActionEvent e) {
    List<String> lst = Arrays.asList(bar);
    foo.peek(40)
}
```



Aside: The *Observer* design pattern

- Applicability
  - When an abstraction has two aspects, one dependent on the other, and you want to reuse each
  - When change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should be able to notify others without knowing who they are
- Consequences
  - Loose coupling between subject and observer, enhancing reuse
  - Support for broadcast communication
  - Notification can lead to further updates, causing a cascade effect



The Java Swing GUI Framework

- Includes a large library of graphical elements
- Monitors system events, dispatches events to appropriate observers
- Updates the screen when graphical elements change

To create a simple Swing application

- Make a window (a JFrame)
- Make a container (a JPanel)
  - Put it in the window
- Add components (JButtons, Boxes, etc.) to the container
  - Use layouts to control positioning
  - Create observers (a.k.a. listeners) to respond to events
- Set up the window to display the container

Swing demos

- The Homework 2 WorldUI
  - An aside for anonymous inner classes…
- My Homework 4 initial dialog

Aside: Anonymous inner classes in Java

- You can implement an interface without naming the implementing class
  - E.g.,

```
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi mom!");
        }
    };

    greeter.run();
}
```

Scope within an anonymous inner class

- An anonymous inner class cannot access non-final variables in the scope where it is defined

```
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    String name = "Charlie";
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi " + name);
        }
    };

    greeter.run();
}
```

compile-time error

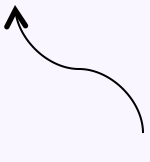
Scope within an anonymous inner class

- An anonymous inner class cannot access non-final variables in the scope where it is defined

```
public interface Runnable {
    public void run();
}

public static void main(String[] args) {
    final String name = "Charlie";
    Runnable greeter = new Runnable() {
        public void run() {
            System.out.println("Hi " + name);
        }
    };

    greeter.run();
}
```



Swing demos

- The Homework 2 WorldUI
  - An aside for anonymous inner classes…
- My Homework 4 initial dialog

Next week

- Midterm exam on Tuesday
- More GUIs on Thursday