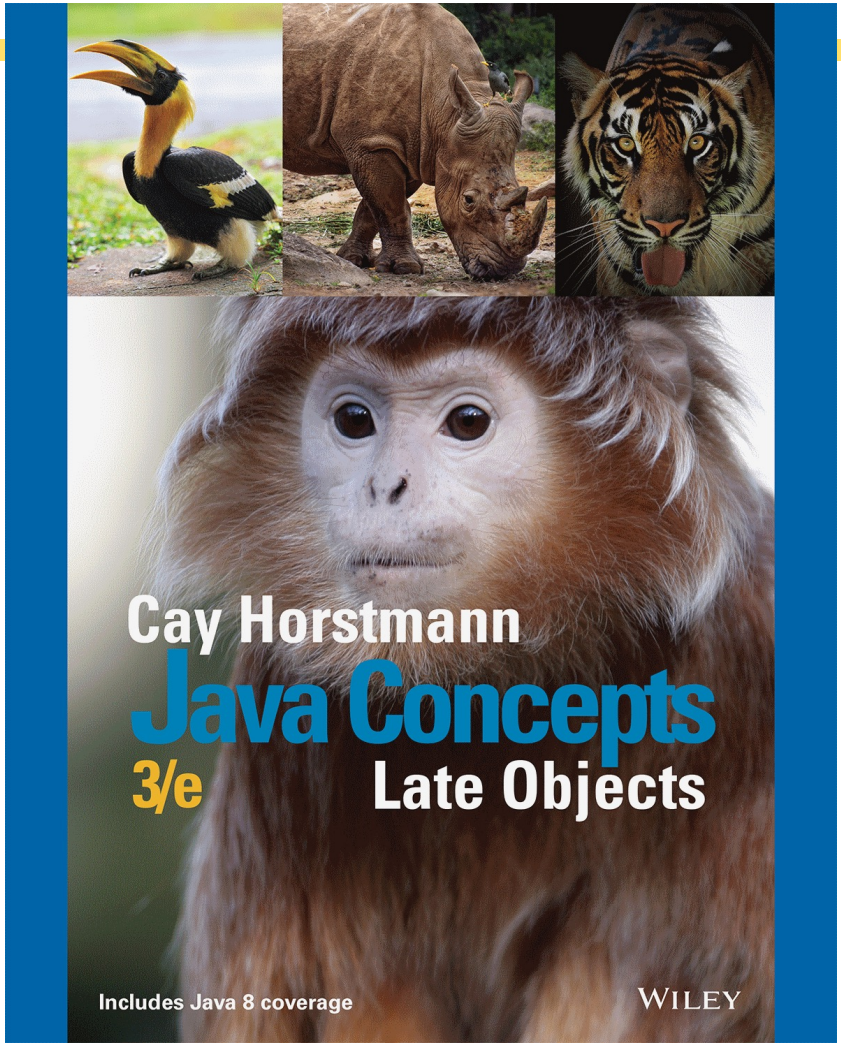
Chapter 15 and 16 - Collections and Generics (Intro)



Announcements

- Final exam on 12:40-2:40 p.m., Mon., December 12.
- Format similar to the midterm. Open book/open notes but compiling and running your code is not permissible.
- No further assignments or quizzes.
- No lab this week on Recursion. Instead we will have an asynchronous lab and possibly a video as instruction.
- Makeup assignment will replace the grade
- 1 quiz will be dropped as an alternative to makeup quiz.

Lesson's Objectives

By the end of this lesson you will:

- Be familiar with the Java collections
- Understand the idea of Java Generics
- Be able to implement sophisticated applications using different Java collections

Collections Overview

Collection classes in Java are containers of Objects which by polymorphism can hold any class that derives from Object (which is actually, any class)

Using Generics the Collection classes can be aware of the types they store

Collections Overview

1st Example:

```
static public void main(String[] args) {
    ArrayList argsList = new ArrayList();
    for(String str : args) {
        argsList.add(str);
    }
    if(argsList.contains("Koko") {
       System.out.println("We have Koko");
    }
    String first = (String)argsList.get(0);
    System.out.println("First: " + first);
}
```

Collections Overview

2nd Example – now with Generics:

```
static public void main(String[] args) {
    ArrayList<String> argsList =
              new ArrayList<String>();
    for(String str : args) {
        argsList.add(str); // argsList.add(7) would fail
    }
    if(argsList.contains("Koko") {
       System.out.println("We have Koko");
    }
    String first = argsList.get(0); // no casting!
    System.out.println("First: " + first);
}
```

Generics

Generics are a way to define which types are allowed in your class or function

```
// old way
List myIntList1 = new LinkedList(); // 1
myIntList1.add(new Integer(0)); // 2
Integer x1 = (Integer) myIntList1.iterator().next(); // 3

// with generics
List<Integer> myIntList2 = new LinkedList<Integer>(); // 1'
myIntList2.add(new Integer(0)); // 2'
Integer x2 = myIntList2.iterator().next(); // 3'
```

Can put here just 0, using autoboxing

Generics

Example 1 – Defining Generic Types:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}

public interface Map<K,V> {
    V put(K key, V value);
}
```

Generics

Example 2 – Defining (our own) Generic Types:

```
public class GenericClass<T> {
    private T obj;
    public void setObj(T t) {obj = t;}
    public T getObj() {return obj;}
    public void print() {
        System.out.println(obj);
    }
}
```

Main:

```
    GenericClass<Integer> g = new GenericClass<Integer>();
    g.setObj(5); // auto-boxing
    int i = g.getObj(); // auto-unboxing
    g.print();
```

Generics – for advanced students

More slides as an appendix

An Overview of the Collections Framework

- A collection groups together elements and allows them to be retrieved later.
- Java collections framework: a hierarchy of interface types and classes for collecting objects.
  - Each interface type is implemented by one or more classes

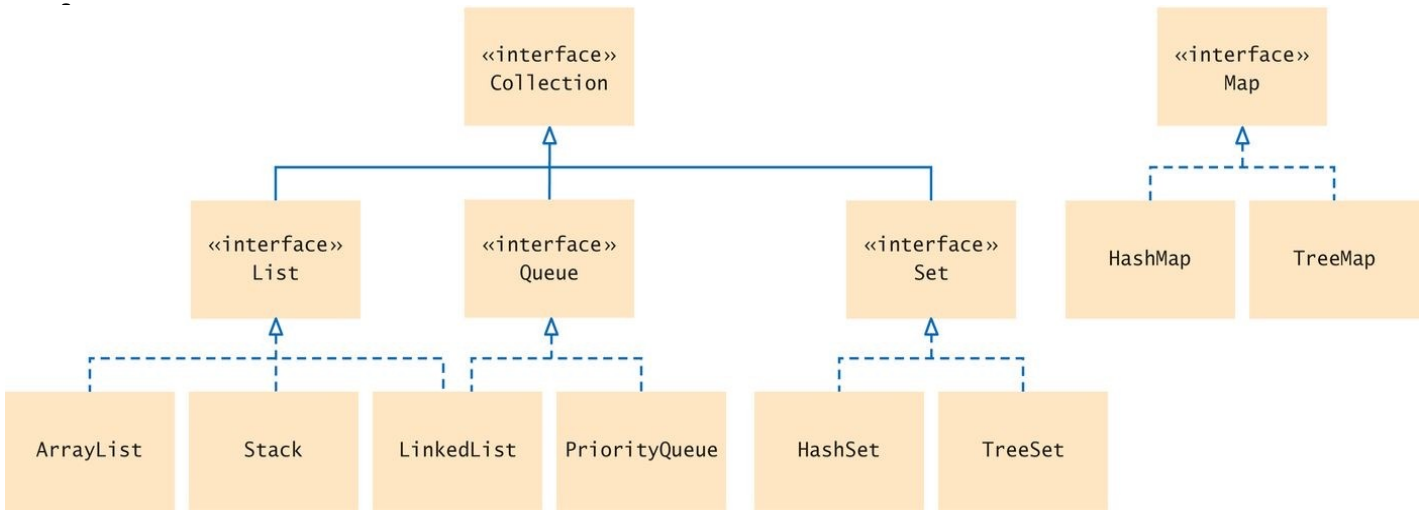


Figure 1 Interfaces and Classes in the Java Collections Framework

- The `Collection` interface is at the root
  - All `Collection` class implement this interface
  - So all have a common set of methods

Vector

Vector is a synchronized dynamically growable array with efficient access by index

Example:

initialCapacity is optional

```
Vector<Integer> vec =
    new Vector<Integer>(10/*initialCapacity*/);
vec.add(7);
```

Vector is an old (Java 1.0) container and is less in use today, replaced mainly by ArrayList (Java 1.2) which is not synchronized

ArrayList

ArrayList is a non-synchronized dynamically growable array with efficient access by index

Example:

initialCapacity is optional

```
ArrayList<Integer> arr =
    new ArrayList<Integer>(10/*initialCapacity*/);
arr.add(7);
```

ArrayList is in fact not a list (though implementing the List interface)
If you need a list use the LinkedList class!

How should I know?

When performing many adds and removes

HashMap

HashMap is a non-synchronized key-value Hashtable

Example 1:

```
HashMap<String, Person> id2Person;
...
Person p = id2Person.get("021212121");
if(p != null) {
   System.out.println("found: " + p);
}
```

HashMap is a Java 1.2 class.
There is a similar Java 1.0 class called Hashtable which is synchronized and is less used today

HashMap

Example 2:

```
HashMap<String, Integer> frequency(String[] names) {
    HashMap<String, Integer> frequency =
              new HashMap<String, Integer>();
    for(String name : names) {
        Integer currentCount = frequency.get(name);
        if(currentCount == null) {
            currentCount = 0; // auto-boxing
        }
        frequency.put(name, ++currentCount);
    }
    return frequency;
}
```

HashMap

Example 2 (cont'):

```
public static void main(String[] args) {
   System.out.println(
      frequency(new String[]{
         "Momo", "Momo", "Koko", "Noa", "Momo", "Koko"
      }).toString());
}
```

HashMap has a nice toString!

Print out of this main is:

{Koko=2, Noa=1, Momo=3}

HashMap doesn't guarantee any order!

HashMap

For a class to properly serve as a key in HashMap the equals and hashCode methods should both be appropriately implemented

Example:

Parameter MUST be Object (and NOT Person!)

```
public class Person {
    public String name;
    boolean equals(Object o) {
        return (o instanceof Person &&
                    ((Person)o).name.equals(name));
    }
    public int hashCode() {
        return name.hashCode();
    }
}
```

Where can this be useful?

Which problem can this be applied to?

Stacks

- A stack lets you insert and remove elements only at one end:
  - Called the top of the stack.
  - Removes items in the opposite order than they were added
  - Last-in, first-out or LIFO order
- Add and remove methods are called `push` and `pop`.
- Example

```
Stack<String> s = new Stack<>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
   System.out.print(s.pop() + " "); // Prints C B A
}
```

- The last pancake that has been added to this stack will be the first one that is consumed.

© John Madden/iStockphoto.

Stacks

- Many applications for stacks in computer science.
- Consider: Undo function of a word processor

  The issued commands are kept in a stack.

  When you select “Undo”, the **last** command is popped off the stack and undone.

- Run-time stack that a processor or virtual machine:

  Stores the values of variables in nested methods.

  When a new method is called, its parameter variables and local variables are pushed onto a stack.

  When the method exits, they are popped off again.

Stack in the Java Library

- `Stack` class provides `push`, `pop` and `peek` methods.

Table 7 Working with Stacks

`Stack<Integer> s = new Stack<>();`	Constructs an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; `s` is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

Queue

- A queue
  - Lets you add items to one end of the queue (the tail)
  - Remove items from the other end of the queue (the head)
  - Items are removed in the same order in which they were added
  - First-in, first-out or FIFO order
- To visualize a queue, think of people lining up.

Photodisc/Punchstock.

- Typical application: a print queue.

Queue

- The `Queue` interface in the standard Java library has:

  an `add` method to add an element to the tail of the queue,

  a `remove` method to remove the head of the queue, and

  a `peek` method to get the head element of the queue without removing it.

- The `LinkedList` class implements the `Queue` interface.
- When you need a queue, initialize a `Queue` variable with a `LinkedList` object:

```
Queue<String> q = new LinkedList<>();
 q.add("A"); q.add("B"); q.add("C");
 while (q.size() > 0) { System.out.print(q.remove() + " "); } // Prints A B  C
```

Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

Backtracking

- Use a stack to remember choices you haven't yet made so that you can backtrack to them.
- Escaping a maze

  You want to escape from a maze.

  You come to an intersection. What should you do?

  Explore one of the paths.

  But remember the other paths.

  If your chosen path doesn't work, you can

  - go back and try one of the other choices.

© Skip ODonnell/iStockphoto.

- Use a stack to remember the paths that still need to be tried.
- The process of returning to a choice point and trying another choice is called **backtracking**.

Backtracking - Maze Example

- Start, at position (3, 4).
- There are four possible paths. We push them all on a stack (1).
- We pop off the topmost one, traveling north from (3, 4).
- Following this path leads to position (1, 4).

  We now push two choices on the stack, going west or east (2).

  Both of them lead to dead ends (3) (4).

- Now we pop off the path from (3,4) going east.

  That too is a dead end (5).

- Next is the path from (3, 4) going south.
- Comes to an intersection at (5, 4).

  Both choices are pushed on the stack (6).

  They both lead to dead ends (7) (8).

- Finally, the path from (3, 4) going west leads to an exit (9).

Backtracking

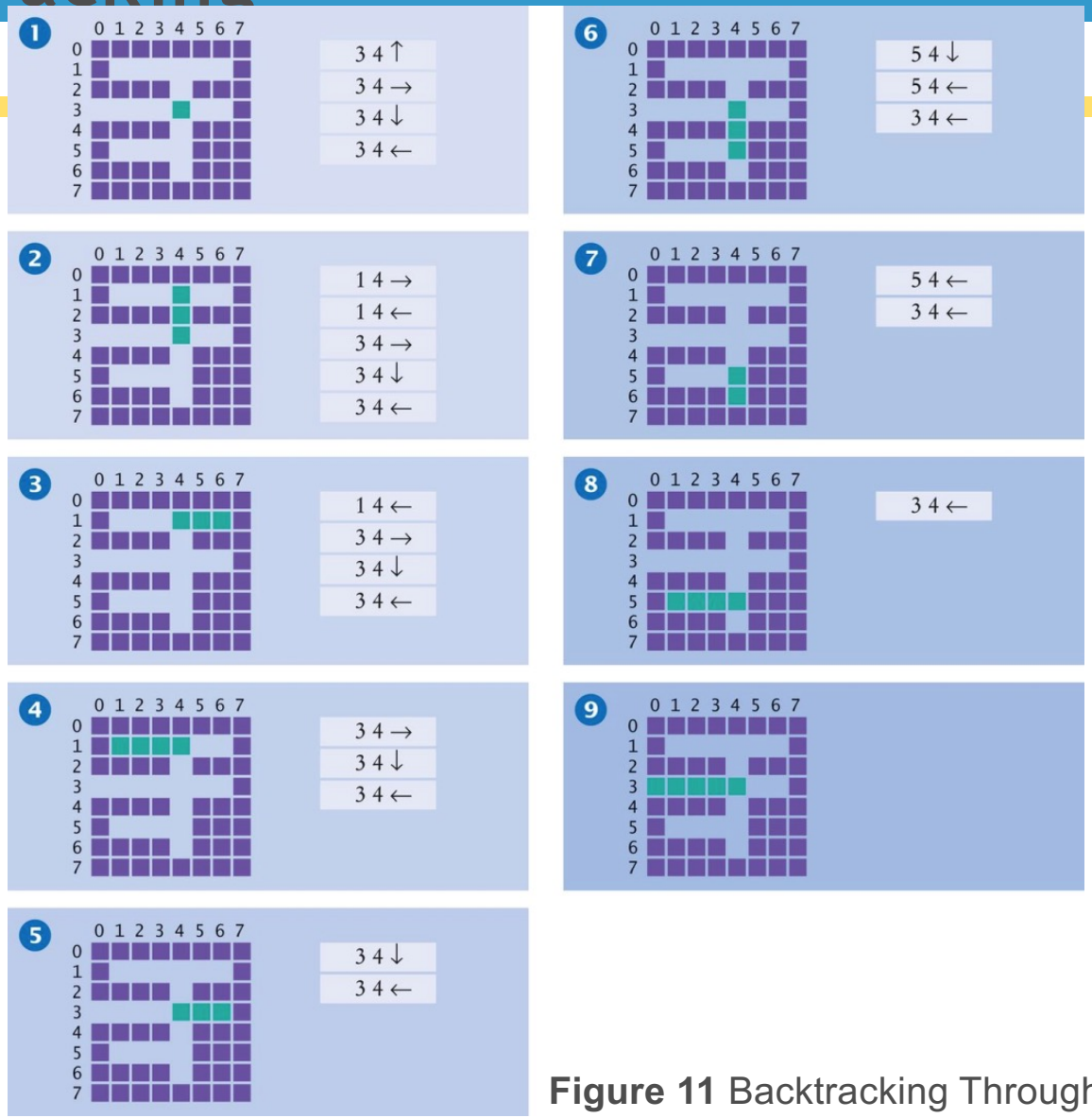


Figure 11 Backtracking Through a Maze

Backtracking

- Algorithm:

```
Push all paths from the point on which you are standing on a stack.
While the stack is not empty
  Pop a path from the stack.
  Follow the path until you reach an exit, intersection, or dead end.
  If you found an exit
    Congratulations!
Else if you found an intersection
  Push all paths meeting at the intersection, except the current one, onto the  stack.
```

- This works if there are no cycles in the maze.

  You never circle back to a previously visited intersection

- You could use a queue instead of a stack.

Collection Utils

Handful Collection utils appears as static methods of the class Collections:

http://java.sun.com/javase/6/docs/api/java/util/Collections.html

A similar set of utils for simple arrays appear in the class Arrays:

http://java.sun.com/javase/6/docs/api/java/util/Arrays.html

What next?

- Android Application Development
- Web programming
- Game development

Many options…

Learn about data structures in more detail and learn how you can use them efficiently to solve complex problems.

All the best for the exam!

Thank you!