CS 407
Design Patterns

Collections Framework & Generics

Dr. Chad Williams
Central Connecticut State University

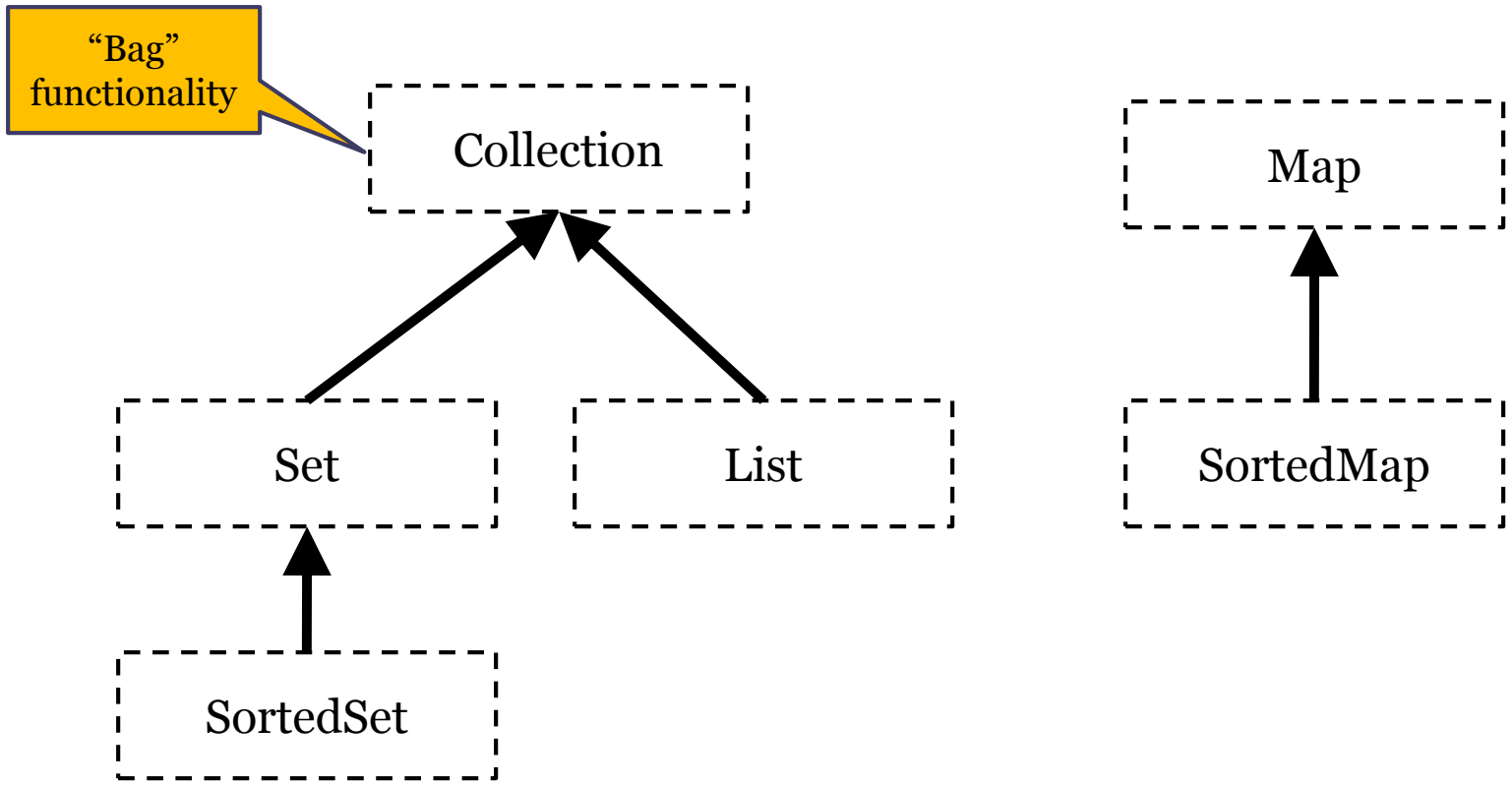
Collections framework

- Set of interfaces and classes that support storing and retrieving data
- Abstraction that allows implementations of various data structures and algorithms to be swapped seamlessly
- Framework is in the `java.util` package

What is it?

- Collection is an object that contains other objects (individual objects referred to as *elements*)
- Based on structures and capabilities collections can classified into major categories known as *abstract collections*
  - Bags, lists, sets, and maps

Abstract collection interfaces



Bags

- Unordered collection of elements
- Can contain duplicates
- Least restrictive form of collection represented by `Collection` interface
- Very limited restrictions, functionality of interface also is very limited

Lists

- `List` ordered collection of elements
- Elements indexed in sequence 0 indexed
- Duplicate elements allowed
- An iterator would return the elements in a repeatable defined way
- Represented as:

  <“object”, ”oriented”, “design”, ”object”>

Sets

- `Set` unordered collection of elements
- **No** duplicates are allowed
  - Inserting an element twice same as inserting it once
- `Set` elements order unrestricted
- Represented as:

  $\{e_1, e_2, \ldots, e_n\}$

  {“object”, ”oriented”, “design”}

SortedSets and sorting

- `SortedSet` extends `Set` interface with restriction the elements are *sort ordered*
- **Ordered** – an iterator would return the elements in a repeatable defined way
- **Sort ordered** – elements are ordered by some type of comparison
  - "Natural" order
  - User defined order through `Comparator`

Maps

- `Map` unordered collection of key-value pairs
- Keys must be unique
- There may be duplication of values
- May retrieve `Set` of keys, `Collection` of values
- Represented as:

  $\{k_1 \text{->} v_1, k_2 \text{->} v_2, \ldots, k_n \text{->} v_n\}$

  {“C”->“carbon”, ”O”->”oxygen”, “He”->”helium”}
- `SortedMap` extends `Map` and requires the set of keys to be sort ordered – returns `Set`, but iterator on Set must be sort ordered

Interfaces

- Refer to `java.util javadoc`

Implementation of collections

- Collections framework provides a set of concrete classes to support each abstract collection
- Developer can choose best implementation for their need from provided classes or build their own
- Within Java all collections are resized dynamically to grow as needed

Implementations for Set

- **HashSet** implements Set – elements stored in a hash table, very efficient insertion and checking if the set contains/lookup up of an element *O(1)*. Unpredictable iteration order
- **LinkedHashSet** implements Set – same implementation as HashSet but an order of the elements is maintained based on insertion order
- **TreeSet** implements SortedSet – elements stored in a red-black binary tree, key set is sorted, insertion and lookup slower *O(log n)*

Implementations for List

- **ArrayList** – elements stored in an array large enough to hold all elements often large portions unused thus extra memory consumed, when array needs to grow its expensive new array allocated all elements copied over, lookup of elements by index *O(1)*.
- **LinkedList** – elements stored in doubly linked list, efficient memory only requires that of actual number of elements, no extra cost for growth, slow lookup of elements by index *O(n)*
- **Vector** – identical to ArrayList but thread safe

Implementations for Map

- **HashMap** – elements stored in a hash table, insert and get are fast *O(1)*, but unordered **Hashtable** is identical but thread safe
- **LinkedHashMap** – similar to HashMap but key order is predictable following insertion order
- **IdentityHashMap** – Same as HashMap but much faster if equality wanted rather than `equals()`
- **TreeMap** implements SortedMap – maintains keys sorted in a red-black binary tree, slower insertion and lookup *O(log n)*

Collections of objects

- Major strength of Collection framework is that any object can be stored in a collection

```
public interface List{
   public boolean add(Object o);
   public Object get(int index);
}
```

Very flexible but also as a result a lot can go wrong with class types and additional work required

Linked list flexibility & weakness

- Result

```
LinkedList list = new LinkedList();
list.add(new Integer(1));
Integer num = (Integer) list.get(0);
```

- Objects of any type added easily but no check to make sure objects are of the same type
- Retrieval of objects always requires a cast
- Enter *Generics* - JSE5 (after the book was published) added the ability to add a type to a generic interface

Generics

- Allow definition of a class to be specified as using a particular class type without knowing that type ahead of time
- Place holder for a class of object which is bound to a specific class at compile time

Idea of generics

- Rather than specifying Object contents let the developer specify contents at compile time
- Classes can be coded with flexibility of accepting any object but developer gains type checking and casting
- Essentially saying List should only hold objects of a specific type

Generics in action

- Pre JSE5.0

```
LinkedList list = new LinkedList();
list.add(new Integer(1));
Integer num = (Integer) list.get(0);
```

- Using Generics

```
LinkedList<Integer> list = new
  LinkedList<Integer>();
list.add(new Integer(1));
Integer num = list.get(0);
```

Generics definitions

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

- The <E> specifies the formal type parameter
- Define all parameters with class/interface name
- Parameters can then be used just like a class name in the rest of the class definition.

What the compiler sees

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}
List<Integer> myList =
             new ArrayList<Integer>();
```

Result:

```
public interface IntegerList{
  void add(Integer x);
  Iterator<Integer> iterator();
}
```

Generics and subtypes

- Rules for generics are a bit different than normal classes for example:

```
String s = "Bob";

Object o = s;
```

- Allowed since Object is a wider version of String, but

```
List<String> sl = new
  ArrayList<String>();

List<Object> ol = sl
```

- Generates a compile time error why?

Generics Example 1

```
static void fromArrayToCollection(Object[] a,
  Collection<?> c) {
   for (Object o : a) {
     c.add(o); // compile time error
   }}
static <T> void fromArrayToCollection(T[] a,
  Collection<T> c) {
   for (T o : a) {
     c.add(o); // correct
}}
```

Generics Example 2

```
static <T> void fromArrayToCollection(T[]
 a, Collection<T> c) {
  for (T o : a) {
    c.add(o); // correct
}}

Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferred to be Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferred to be String
fromArrayToCollection(sa, co);// T inferred to be Object
```

Generics example 3

```
Integer[] ia =  new Integer[100];
Float[] fa =    new Float[100];
Number[] na =   new Number[100];
Collection<Object> co = new ArrayList<Object>();
Collection<Number> cn =   new ArrayList<Number>();
Collection<String> cs =   new ArrayList<String>();


fromArrayToCollection(ia, cn);// T inferred to be Number
fromArrayToCollection(fa, cn);// T inferred to be Number
fromArrayToCollection(na, cn);// T inferred to be Number
fromArrayToCollection(na, co);// T inferred to be Object
fromArrayToCollection(na, cs);// compile-time error
```

Wildcards

```
void printCollection(Collection<Object> c)
  {
    for (Object e : c) {
       System.out.println(e);
    }
}
```

- Would work only with a collection typed to Objects any other type couldn't use this method
- Wild card character "?" matches any type of typed class

```
void printCollection(Collection<?> c) {
   for (Object e : c) {
     System.out.println(e);
   }
}
```

Wild card restrictions

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
    System.out.println(e);
  }
}
```

- Allows you to read out of any collection
- Can't add to the collection as its not able to bind to the collection type at compile time

```
Collection<?> myCollection = new ArrayList<String>();
myCollection.add("foo");
```

- Results in compile error

Bounded wild cards

- Class Manager extends Employee

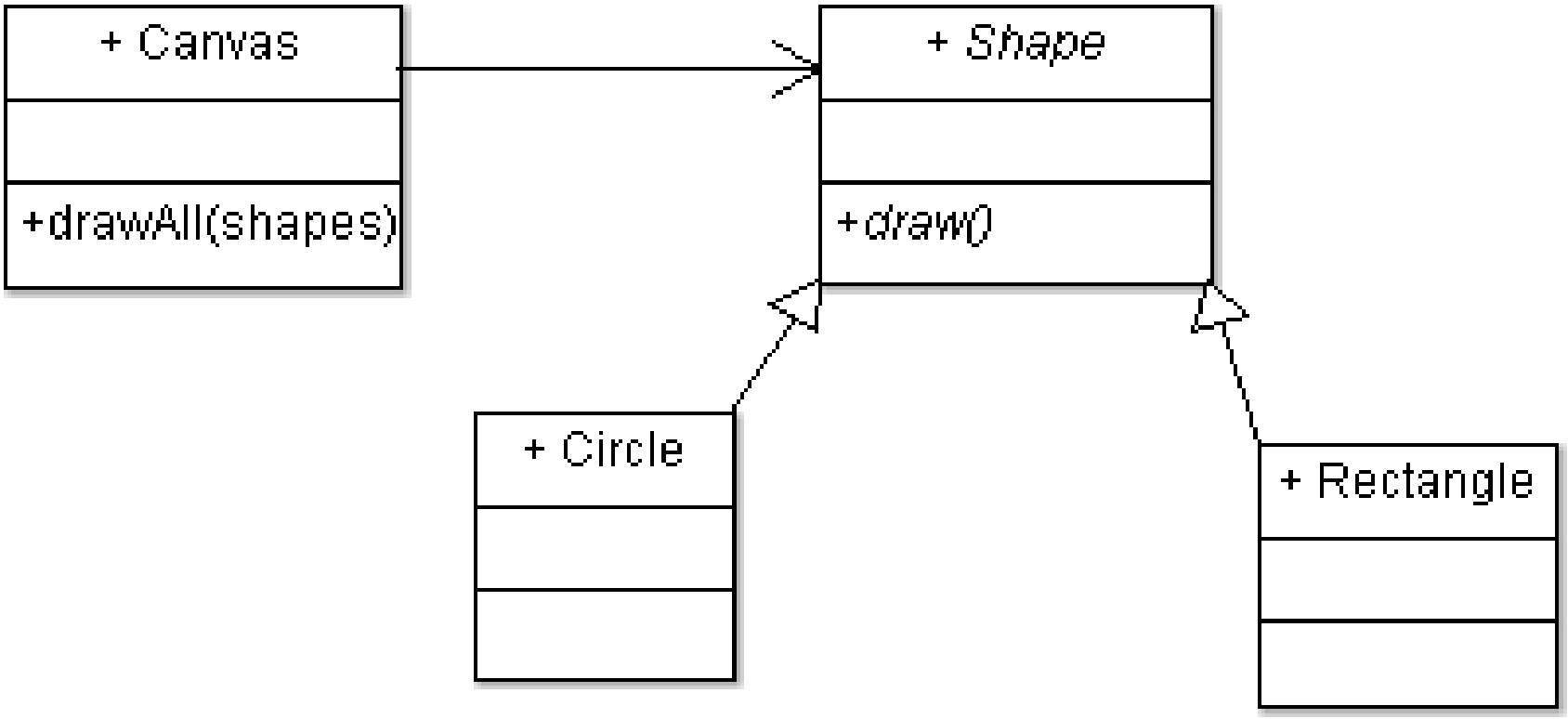
```
public void printInfo(Collection<Employee> l);
```

- Would like to be able to pass in a Collection<Manager>, solution bounded wild card

```
public void printInfo(
        Collection<? extends Employee> l);
```

- Like a basic wildcard though can only read from collection, add would cause a compile time error

Bounded wild card example



Group work - generics

- Figure out the method signatures

```
public interface Map<K,V>{
   put()     // insert new
   get()     // retrieve(note takes object)
   keySet() // returns Set of keys
   values() // returns Collection of values
}
```

More generics examples

- Class definition **`WidgetCollection`**, on creation it is bound to a specific class. It should have a **list of the bound widgets** on the class and have a method to **add** a new widget of that class and get an **iterator** of widgets of that class

Group work

- 2 classes Employee and Manager which is a subclass of Employee – Employee has a method getName() that returns a String

- Write a function that takes a name and a List of Employees or Managers and returns a List of Employees with that first name

Final example

- ***This one is a bit fancy, but give it a try:*** Managers can oversee Employees or any of its specific subclasses, but if bound to a subclass that Manager can only oversee employees of that subclass. (i.e. if bound to Managers can only oversee managers).

Add a List to the Manager named `manages` that contains classes of the type specified at compile. Add a method to add new objects of this type to the list.

Group work - generics solution

- Figure out the method signatures

```
public interface Map<K,V>{
   put(K key, V value) // insert new
   V get(Object key)   // retrieve
   Set<K> keySet()  // Set of keys
   Collection<V> values()  // values
}
```

Solution

Class definition `WidgetCollection`, on creation it is bound to a specific class. It should have a list of the bound widgets on the class and have a method to add a new widget of that class and get an iterator of widgets of that class

```
public class WidgetCollection <T> {
  List<T> widgets;
  public WidgetCollection (){
    widgets = new ArrayList<T>();
  }
  public void add (T widget){}
  public Iterator<T> iterator(){}
}
```

Solution

Write a function that takes a name and a List of Employees or Managers and returns a List of Employees with that first name

```
List<Employee> getMatching(String name,
  List<? extends Employee> employees){
 List<Employee> matches = new ArrayList<Employee>();
 for (e : employees){
   if (e.contains(name)){
     matches.add(e);
   }
 }
 return matches;
}
```

Solution

Managers can oversee Employees or any of its specific subclasses, but if a subclass that Manager can only oversee employees of that subclass. Add a List to the Manager named `manages` that contains classes of the type specified at compile. Add a method to add new objects of this type to the list.

```
public class Manager<T extends Employee> extends
Employee{
  List<T> manages;
  public Manager(){
    manages = new ArrayList<T>();
  }
  public void addManagee(T person){
    manages.add(person);
  }
}
```