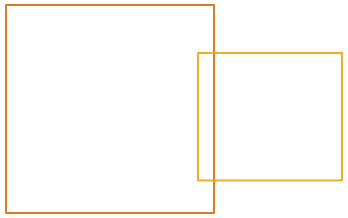
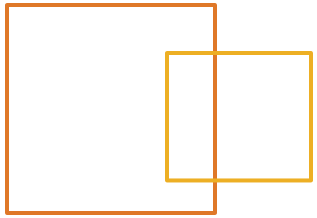
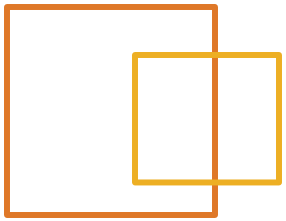
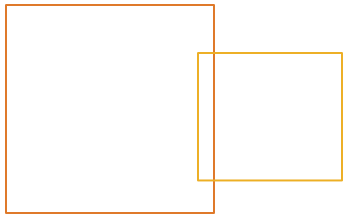
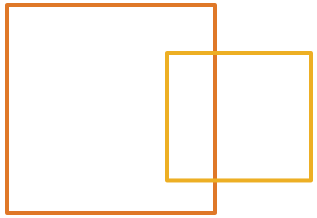
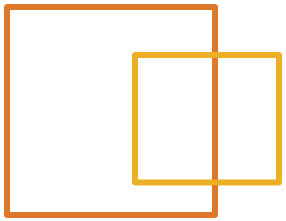





Fast Track to Java

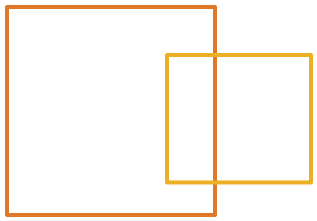
Customized for Starbucks

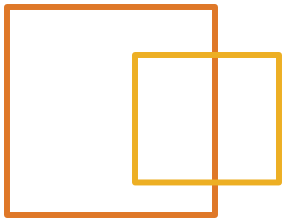
Delivered by DevelopIntelligence


Collections

Objectives

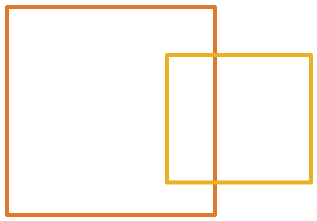


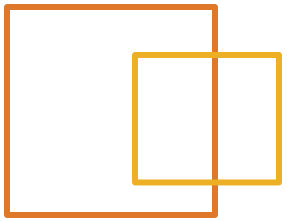


At the end of this module you should be able to

- Describe the Collections Framework architecture
- Use an `Iterator`
- Use a `Set`
- Use a `List`
- Use a `Map`
- Use collection algorithms
- Use wrappers

Collections



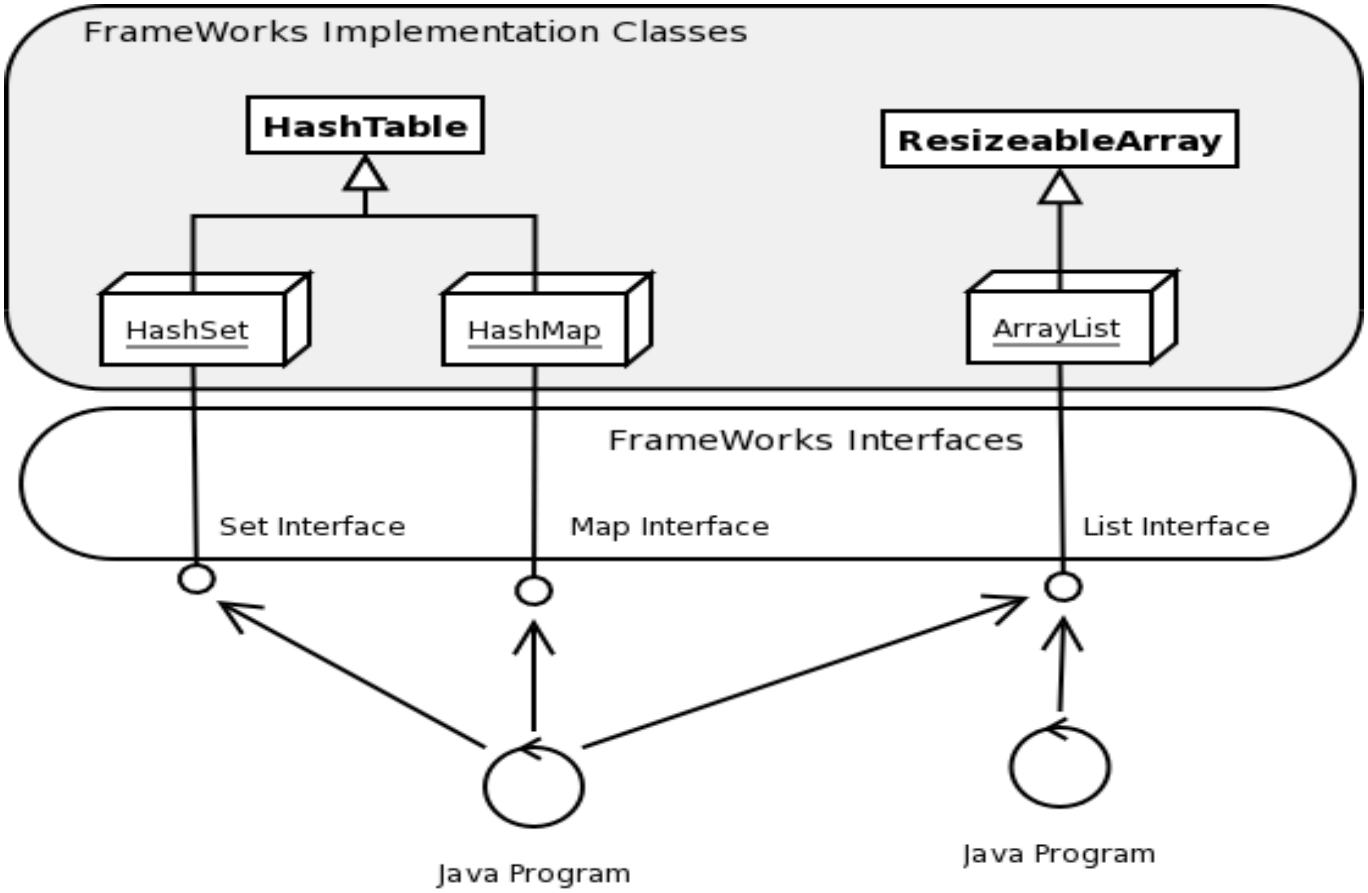


- A collection is a container for other objects
- Arrays are a basic type of collection
- Java provides several collection types, e.g.:
  - Bag
  - List
  - Set
  - Map

Collections Framework

- Collections API has three key elements
  - Interfaces
    - Expose the functionality of collections
    - Underlying container is manipulated through the interface
    - Client is not coded to the implementation
    - Trivializes changing implementations
  - Implementations
    - The data structure mechanisms themselves
    - Possibly add more, specific, functionality
  - Algorithms and Wrappers
    - Reusable external functionality
    - Sorting and searching

The Java Collections Framework Architecture



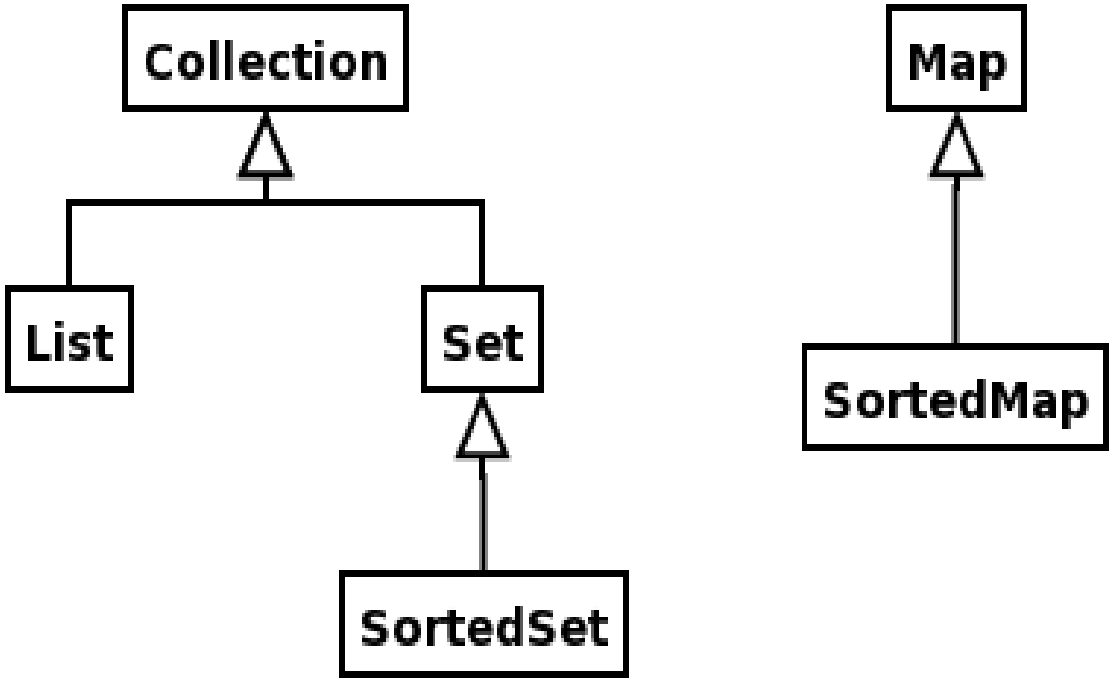
Part of the Collections Framework architecture

Collection Types

Two main categories of collections

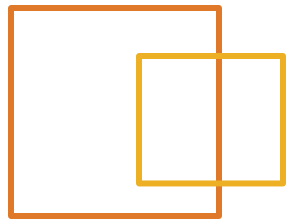
- `java.util.Collection`
  - Root interface in the *collection hierarchy*
  - May contain duplicates
  - May be ordered
  - Useful only through implementations like
    - `ArrayList`
    - `HashSet`
- `java.util.Map`
  - An object that maps keys to values
  - Cannot contain duplicate keys
  - Each key can map to at most one value
  - Useful only through implementations
    - `TreeMap`
    - `HashMap`

The Collections Interfaces



Frameworks Interface Hierarchy

Collection Interface API



```
public interface Collection {
    // Basic Operations
     int size();
     boolean isEmpty();
     boolean contains(Object element);
     boolean add(Object element);     // Optional
     boolean remove(Object element);  // Optional
     Iterator iterator();

    // Bulk Operations
     boolean containsAll(Collection c);
     boolean addAll(Collection c);        // Optional
     boolean removeAll(Collection c);     // Optional
     boolean retainAll(Collection c);     // Optional
     void clear();                        // Optional

    // Array Operations
     Object[] toArray();
     Object[] toArray(Object a[]);
}
```

Using the Collections Framework

Basic steps for using collections framework

1. Select the interface appropriate for the application
2. Select the desired data structure implementation
3. Instantiate the implementation
4. Manipulate the data structure using the interface

Creating, Filling & Printing Collections Example

```
import java.util.*;
// This is a utility class that provides a method for
//  filling a collection -- any collection because it only uses
//  the methods in the collection interface. This shows the
//  use of the Collections type as a general type for passing
// as an argument.
class Fill {
  static Collection init(Collection c, int slots) {
    for (int i = 0; i < slots; i++) {
      c.add("Test Value " + i);
    }
    return c;
  }
}
```

Creating, Filling and Printing Collections (cont.)

```
public class UseSomeCollections {
  public static void main(String[] args) {
    Collection arrayList = new ArrayList();
    Collection hashSet = new HashSet();
    Collection treeSet = new TreeSet();
    Collection linkList = new LinkedList();
    arrayList = Fill.init(arrayList,5);
    hashSet = Fill.init(hashSet,5);
    treeSet = Fill.init(treeSet,5);
    linkList = Fill.init(linkList,5);
    System.out.println("ArrayList");
    System.out.println(arrayList);
    System.out.println("HashSet");
    System.out.println(hashSet);
    System.out.println("TreeSet");
    System.out.println(treeSet);
    System.out.println("LinkedList");
    System.out.println(linkList);
  }
}
```

Creating, Filling and Printing Collections Output

```
// Output is
ArrayList
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
HashSet
[Test Value 2, Test Value 3, Test Value 1, Test Value 0, Test Value 4]
TreeSet
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
LinkedList
[Test Value 0, Test Value 1, Test Value 2, Test Value 3, Test Value 4]
```

Iterator Interface API

- Both java.util.Collection and java.util.Map provide a mechanism to iterate over the contained values
- Iterator is an interface describing how to Iterator over the collection
- Each implementation class will provide its own Iterator implementation

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```

Iteration Example

```
import java.util.*;
// Now we have added a generic Iterator method
class Fill {
   static Collection init(Collection c, int slots) {
     for (int i = 0; i < slots; i++) {
       c.add("Test Value " + i);
     }
     return c;
   }
   static void deleteSecond(Collection c) {
     Iterator itr = c.iterator();
     boolean even = false;
     while (itr.hasNext()) {
       itr.next();
       if (even) {
        itr.remove();
       }
       even = !even;
     }
   }
}
```

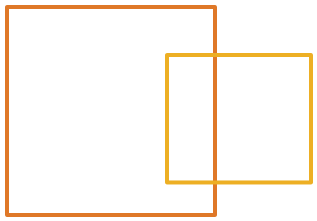
Iteration Example (cont.)

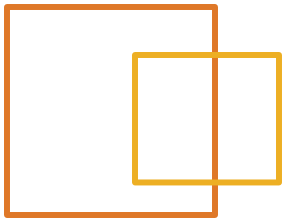
```
public static void main(String[] args) {
  Collection arrayList = new ArrayList();
  Collection hashSet = new HashSet();
  Collection treeSet = new TreeSet();
  Collection linkList = new LinkedList();
  arrayList = Fill.init(arrayList, 5);
  hashSet = Fill.init(hashSet, 5);
  treeSet = Fill.init(treeSet, 5);
  linkList = Fill.init(linkList, 5);
  System.out.println("ArrayList");
  Fill.deleteSecond(arrayList);
  System.out.println(arrayList);
  System.out.println("HashSet");
  Fill.deleteSecond(hashSet);
  System.out.println(hashSet);
  System.out.println("TreeSet");
  Fill.deleteSecond(treeSet);
  System.out.println(treeSet);
  System.out.println("LinkedList");
  Fill.deleteSecond(linkList);
  System.out.println(linkList);
}
```

Iteration Example Output

```
// Output is
ArrayList
[Test Value 0, Test Value 2, Test Value 4]
HashSet
[Test Value 2, Test Value 1, Test Value 4]
TreeSet
[Test Value 0, Test Value 2, Test Value 4]
LinkedList
[Test Value 0, Test Value 2, Test Value 4]
```

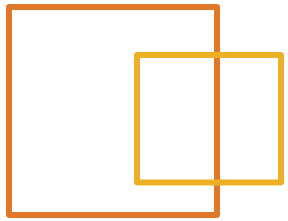
Set Interface





- A set is a collection that contains no duplicates
- A sub-interface of `java.util.Collection`

Set Interface Example



```
import java.util.*;
class Test{}  // something to put in the Set
public class TestASet {
  public static void main(String [] args) {
    Set s = new HashSet();  // create the set
    Test t = new Test();
    s.add(t);
    s.add(t);     // duplicate entry
    s.add("One");
    s.add("Two");
    s.add("One");
    s.add("One");
    s.add("Three");
    s.add("Four");
    s.add("Four");
    s.add("Four");
    s.add(new Test());  /// not a duplicate
    System.out.println(s);
  }
}
// Output is:
[Test@107077e, Test@11a698a, Four, Three, Two, One]
```

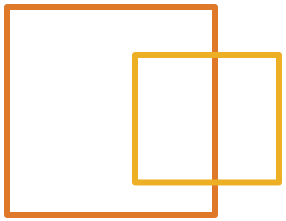
List Interface

- An ordered collection, or sequence
- A sub-interface of `java.util.Collection`
- May contain duplicate elements
- Implementations typically allow `null`
- Supports positional access for insertion and retrieval (based on index)
- Has a special type of `Iterator`, `ListIterator`
  - Allows insertion and replacement while iterating over the collection
  - Supports `Iterator` interface operations

List Interface API

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);           // Optional
    void add(int index, Object element);             // Optional
    Object remove(int index);                        // Optional
    abstract boolean addAll(int index, Collection c); // Optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

List Iterator API



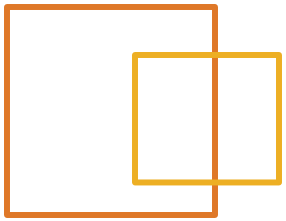
```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();
    boolean hasPrevious();
    Object previous();
    int nextIndex();
    int previousIndex();

    void remove();          // Optional
    void set(Object o);     // Optional
    void add(Object o);     // Optional
}
```

List Example

```
import java.util.*;
public class TestAList {
  public static void main(String[] args) {
    List L = new LinkedList();
    for (int i = 0; i < 10; i++) {
      L.add("" + i);
    }
    System.out.println("List created");
    System.out.println(L);
    L.add(4, "10");
    System.out.println(L);
    L.set(5, "11");
    System.out.println(L);
    ListIterator itl = L.listIterator(4);
    System.out.println("L[4]=" + L.get(4));
    itl.previous();
    itl.remove();
    System.out.println(L);}
  }
}
```

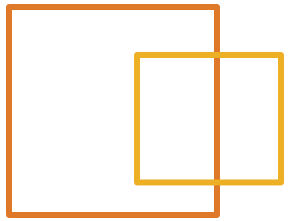
```
// output
List created
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 10, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 10, 11, 5, 6, 7, 8, 9]
L[4]=10
[0, 1, 2, 10, 11, 5, 6, 7, 8, 9]
```

Map Interface



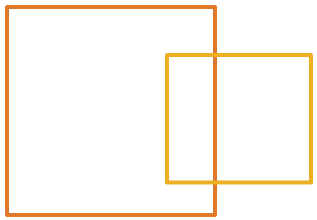
- Maps keys to values
  - Like a micro-database, two columns, key and data
- Contains no duplicate keys, values may be duplicates
- No direct Iterator functionality
- Provides three views of data that allow us to obtain Iterators
  - Keys
  - Values
  - Entry set (key-value mappings)

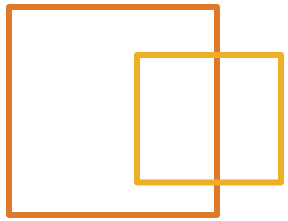
Map Interface API



```
public interface Map {
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    void putAll(Map t);
    void clear();

    public Set keySet();
    public Collection values();
    public Set entrySet();

    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

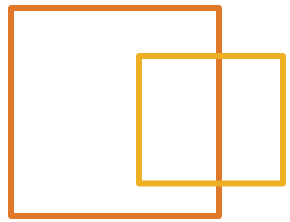
Map Example





```
public class TestAMap {
  public static void main(String[] args) {
    Map custs = new HashMap();
    custs.put("982098", new Customer("Bill White"));
    custs.put("116201", new Customer("Bob Green"));
    custs.put("983611", new Customer("Saj Black"));
    custs.put("661109", new Customer("Sharon Brown"));
    System.out.println(custs);

    custs.remove("116201");
    custs.put("761102", new Customer("Simone Blanc"));
    System.out.println(custs.get("661109"));
    System.out.println(custs);
  . . .
```

Map Example (cont.)

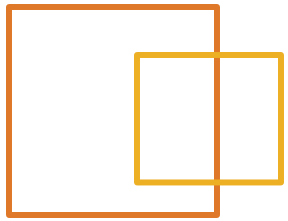


```
 // Now walk through the entries
 Set entries = custs.entrySet();
 Iterator iter = entries.iterator();
 while (iter.hasNext()) {
   Map.Entry entry = (Map.Entry) iter.next();
   Object key = entry.getKey();
   Object value = entry.getValue();
   System.out.println("key=" + key + ", value=" + value);
 }
} //end main
} //end class
```

java.util.Collections

- A utility class that provides
  - Algorithms
  - Wrappers
- Static methods for common algorithms for things like
  - Binary search
  - Reversing
  - Shuffling
  - Sorting
- Wrappers for creating
  - Singletons
  - Synchronized collections
  - Unmodifiable collections
- See also `java.util.Arrays` class

Collections Example

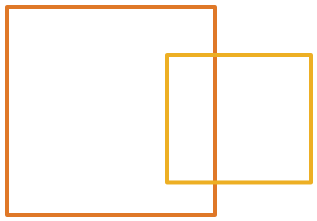


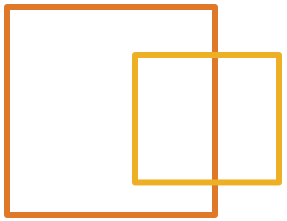
```
public class TestCollectionsUtils {
    public static void main(String[] args) {
        List numbers = new ArrayList(12);
        for (int i = 1; i <= 12; i++) {
            numbers.add(new Integer(i));
        }
        System.out.println("Starting List\n" + numbers);

        Collections.shuffle(numbers); // Randomize
        System.out.println("Shuffled List\n" + numbers);

        Collections.sort(numbers); // Sort
        System.out.println("Sorted List\n" + numbers);

        numbers = Collections.unmodifiableList(numbers);
        Collections.shuffle(numbers); // woops!
    }
}
```

Collections Example (cont.)

```
Starting List
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Shuffled List
[7, 10, 4, 1, 9, 11, 12, 8, 5, 2, 3, 6]
Sorted List
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Exception in thread "main" java.lang.UnsupportedOperationException
 at java.util.Collections$UnmodifiableList.set(Collections.java:1156)
[...]
 at tests.TestCollectionsUtils.main(TestCollectionsUtils.java:24)
```

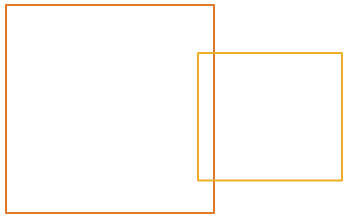
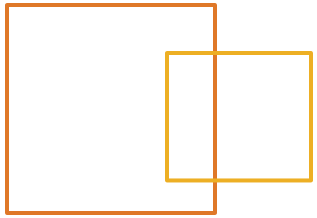
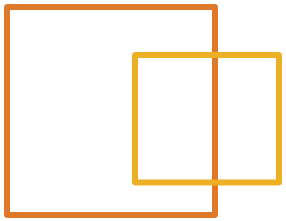
Summary





We covered

- Describing the Collections Framework architecture
- Using an `Iterator`
- Using a `Set`
- Using a `List`
- Using a `Map`
- Using an algorithm
- Using wrappers


Generics

Black and white never tasted so good

Objectives

At the end of this module you should be able to:

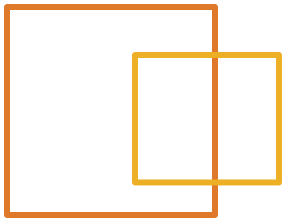
- Use generic collections
- Understand the basic principles of generics
- Recognize and use generic methods and classes
- Understand why generics and polymorphism causes design complexity

Generics

What are generics?

- Stands for generic types and generic methods
- Represent design pattern known as *parameterized types* and methods
- Allows a type to be defined without specifying all of the other types it uses
- Were one of most requested features of language

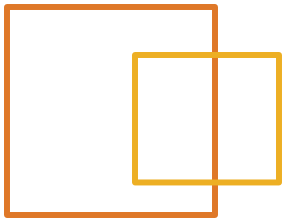
Generics [cont.]



Why do they exist?

- Add type-awareness to collections, without breaking flexibility
- Add type-awareness to other container-like classes, without breaking flexibility
- Add type awareness to methods
- Provide compile-time type-safety
  - Remove development-time casting procedures
  - Remove run-time type incompatibilities
  - Remove run-time `ClassCastExceptions`

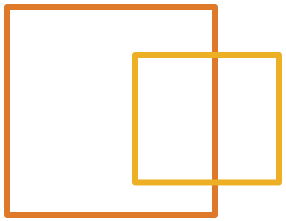
Using Generics



How do they work?

- Supports both definition and application
  - Most straightforward is application
  - But application requires understanding definition
- Use “placeholder” to represent generic type as part of type or method definition
  - Placeholder value is replaced with type in source
  - Placeholder is removed during compilation, replaced with traditional casting (known as type erasure)

Generic Placeholders

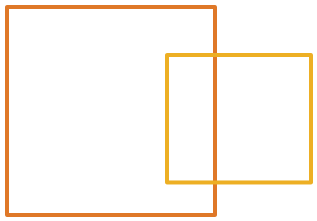


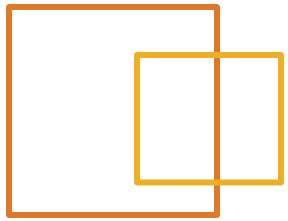
- Generic type placeholders
  - Used when defining a parameterized type
  - <E> - stands for element; represents element type held within container
  - <T> - stands for type
  - <V> - stand for value
- Generic method placeholders
  - Used when defining a parameterized method
  - `<E>` - parameterized type
  - `<?>` - wildcard placeholder
  - `<? extends E>` - bounded wildcard placeholder
  - `<? super E>` - bounded wildcard placeholder
- `<E>, <T>, <V>,` etc. naming convention only

Generic Collections

- Collections API has been rewritten to support Generics
  - Provides type safety to collections
  - Applies to *all* classes within Collection API
- Specify the type the collection will hold
  - Inserting type mismatch generates compile-time error
  - Getting / removing element no longer requires cast
- Backwards compatible in *raw type* format
  - May generate compile-time warning
  - Can widen typed collection into raw-type

Generic List





- `List` represents an ordered collection
- `List` interface now represents generic type

```
public interface List<E> extends Collection<E> {..}
```

- Read as *List of <type E> elements*
- Certain `List` methods now generic
  - `Iterator<E> iterator();`
  - `boolean containsAll(Collection<?> c);`
  - `boolean addAll(Collection<? extends E> c);`

Generic ArrayList

- Provides type-safe representation of an array-backed list
  - Implementation of `List` interface
  - Subclass of `AbstractList`
  - Common replacement for `Vector`
  -
- Create `ArrayList` using parameterized syntax:
  - `List<String> myList = new ArrayList<String>();`
  - `<String>` replaces placeholder `<E>`
  - Read as *`List` of `String` elements*
  - `myList` can only hold `String` elements

Simple List Example [Old way]

```
1   package examples.generics.simple;
2   import ...
6   /**...*/
11  public class OldWayExample {
12
13    public static void main(String[] args) {
14      List myList = new ArrayList();
15      //convert args into a List
16      List argList = Arrays.asList(args);
17      //add Strings to list
18      myList.addAll(argList);
19      //list is not typesafe, can add any object
20      myList.add(new Integer(0));
21
22      Iterator theArgs = myList.iterator();
23
24      //step through list elements
25      while (theArgs.hasNext()) {
26        //will cause class cast
27        // exception with Integer element
28        String nextArg = (String) theArgs.next();
29      }
30    }
31  }
32
```

Simple List Example [New way]

```
1   package examples.generics.simple;
2   import ...
6   /**...*/
10  public class TestExample {
11
12    public static void main(String[] args) {
13      //typesafe List of String elements
14      List<String> myList = new ArrayList<String>();
15
16      //convert args into a List<String>
17      List<String> argList = Arrays.asList(args);
18      myList.addAll(argList);
19
20      //would cause compile-time error
21      //myList.add(new Integer(0));
22
23      //Iterator is now also typesafe
24      Iterator<String> theArgs = myList.iterator();
25      while(theArgs.hasNext()) {
26        String nextArg = theArgs.next();
27      }
28    }
29  }
30
```

Typesafe Collection Advantages

- Adds compile time type safety
  - `OldWayExample` allowed `Integer` to be inserted into collection; discovered problem at run-time
  - `TestExample` prevented `Integer` to be inserted into collection; discovered at compile-time
- Simplified interactions
  - `OldWayExample` required casting when working with collection elements
  - `TestExample` contained specific type; so no casting needed
- No advantages in speed or performance

How Do They Work? [revised]

Implemented differently from other languages

- Adopt ***type erasure*** mechanism
  - Parameterized placeholder replaced at compile time
  - Code converted from parameterized to generic
  - Compiler “inserts” cast similar to `OldWayExample`
  -
- Compiler ensures type-safety
  - Only at compile time
  - Run-time relies on traditional mechanism
  - As a result, can still encounter run-time exceptions

Simple List Example [corrupted]

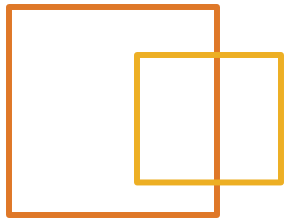
- Third-party API does not utilize type-safe collections
- Causes issues at run-time (adds an Integer)

```
1   package examples.generics.simple;
2
3   import java.util.List;
4
5   /**...*/
11  public class ThirdPartyAPI {
12
13    public static void addElement(List list) {
14      list.add(new Integer(32));
15    }
16  }
17
```

Simple List Example [corrupted]

```
1   package examples.generics.simple;
2   import ...
6   /**...*/
10  public class CorruptTestExample {
11
12    public static void main(String[] args) {
13      //typesafe List of String elements
14      List<String> myList = new ArrayList<String>();
15
16      //convert args into a List<String>
17      List<String> argList = Arrays.asList(args);
18      myList.addAll(argList);
19
20      //call third-party api which uses raw types
21      ThirdPartyAPI.addElement(myList);
22
23      //Iterator is now also typesafe
24      Iterator<String> theArgs = myList.iterator();
25      while(theArgs.hasNext()) {
26        String nextArg = theArgs.next();
27      }
28    }
29  }
30
```

Solidifying Type-safety



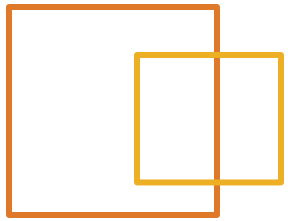
- Type erasure can be cheated
- Want a facility to enforce type-safety
- Collection class adds type-checked wrappers
  - Prevents insertion of type mismatched objects
  - Encounter `ClassCastException` on inappropriate insertion
  - Easier to debug—fails faster

Solidifying Type-safety [cont.]

- Dynamic type-safety support provided by collections class
  - `Collections` class rewritten to support generics
  - New static methods used to create a “checked” collection

```
public static <E> List<E> checkedList(
        List<E> list, Class<E> type);
public static <K, V> Map<K, V> checkedMap(
        Map<K, V> m, Class<K> keyType,

        Class<V> valueType);
```

Working with Generics



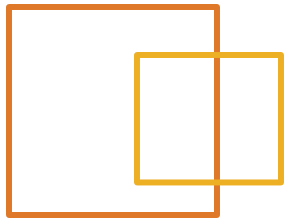
- Can create generic methods
- Can create own generic types

Creating Generic Methods

Relatively straightforward process

- Can add generic method support to any class
- Use when you want to place type constraints on method
- Simply add generic method nomenclature to method signature
  - Declare generic types
  - Adjust method parameter list
  - Adjust method return signature

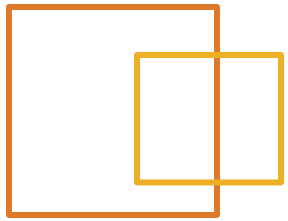
Generic Method Example



- Can create method with signature to ensure compile-time type-safety

```
public class MyGenericMethod {
  public static <T> List<T> getAList(Class<T> type, T... item){
    List<T> theList =
      Collections.checkedList(new LinkedList<T>(), type);
    theList.addAll(Arrays.asList(item));
// equivalent to:
//  for (T anItem : item) {
//    theList.add(anItem);
//  }
    return theList;
}
```

Generic Method Example

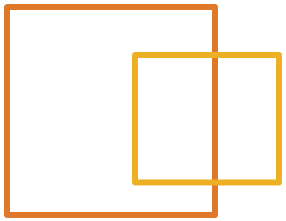


```
public static void main(String[] args) {
  // Create a list of String
  List<String> sl = getAList(String.class,
    "Hello", "There", "How", "Are", "You?");
    // Create a list of numbers
    List<Number> nl = getAList(Number.class, 1, 3, 9.2);

    System.out.println("Strings: " + sl);
    System.out.println("Numbers: " + nl);
    // insert wrong type?
    tryToBreakIt(nl);
}

public static void tryToBreakIt(List l) {
  l.add("Surely not?");
}
```

Generics and Polymorphism

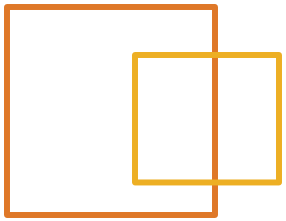
- How are `List<Account>` and `List<BankAccount>` related?
- `BankAccount` Is-A `Account`
  - So, `processAccount(Account a)` can take `BankAccount` as a parameter
- But `List<BankAccount>` is ***not*** `List<Account>`
  - Otherwise, it would be possible to add an `Account` to a `List<BankAccount>`, which would be bad
- So, `processAccounts(List<Account> la)` cannot take `List<BankAccount>` as an argument
  - Life gets complicated very fast!

Wildcard Generic Types



- Wildcards allow flexibility in method signature
  - `<?>` - unknown type
  - unbounded wildcard
  - use when you don't know or care about the value's type; like raw types
  - `<? extends Number>`
  - Upper-bounded wildcard
  - Specified type should be Number or any subclass of Number
  - `<? super Number>`
  - Lower-bounded wildcard
  - Specified type should be a subclass of Number
- Wildcards might limit read or write access

Creating Generic Types



- Relatively straight-forward process
  - Create generic type like any type
  - Include generic type nomenclature
  - Reference placeholder within code
  - Have methods support generic type
  -
- Could be used for things like:
  - Custom data structure
  - Generic value object

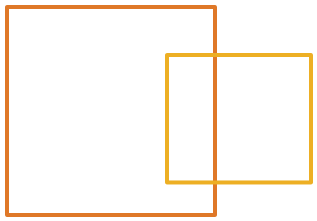
GenericVO Example

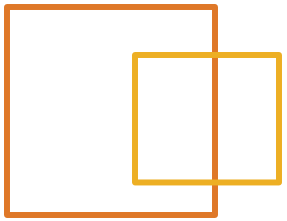
```
1   package examples.generics.advanced;
2
3   public class GenericVO<A,B> {
4
5     A fieldA;
6     B fieldB;
7
8     GenericVO(A a, B b) {
9       fieldA = a;
10      fieldB = b;
11    }
12
13    public void setFieldA(A a) {
14      this.fieldA = a;
15    }
16
17    public A getFieldA() {
18      return fieldA;
19    }
20
21    public void setFieldB(B b) {...}
24    public B getFieldB() {...}
27  }
28
```

GenericVO Example [cont.]

```
1  package examples.generics.advanced;
2
3  public class GenericVOExample {
4
5    public static void main(String[] args) {
6      //create instances of the GenericVO
7      GenericVO<String, String> name =
8              new GenericVO<String,String>("John", "Doe");
9
10     GenericVO<String, Integer> user =
11             new GenericVO<String,Integer>("john_doe123", 123457);
12
13     //get name field <B>
14     String lastName = name.getFieldB();
15     System.out.println("name's field <B> is: " + lastName);
16
17     //get user field <B>
18     Integer userId = user.getFieldB();
19     System.out.println("users's field <B> is: " + userId);
20   }
21
22 }
23
```

Summary

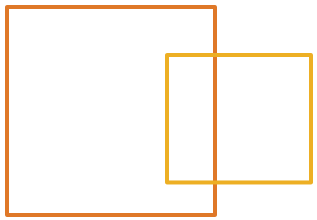


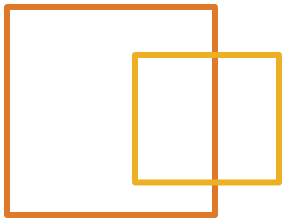


In this module, we covered:

- Using generic collections
- The basic principles of generics
- Recognizing and using generic methods and classes
- Why generics and polymorphism cause design complexity

Lab 8





- Description: Use the Mixer class in the **MixerLab** project as your starting point. Refactor the Mixer so that the frequency map becomes type safe. The frequency map should contain <String, Integer> as its map structure. The List should contain <String> as its element types.
- You should get rid of all the “raw type” warnings that Eclipse gives you.
- Duration: 15 minutes

Lab 8.5 (optional)

- Description: The `getFrequencyMap` in the `Mixer` class is very inefficient as written:
  - Why is that?
  - Rewrite it to be able to create the Map in one iteration through the `args` array.
- Duration: 15 minutes