Generics
ArrayLists and arrays

- A **ArrayList** is like an array of **Objects**, but...
  - Arrays use `[ ]` syntax; **ArrayLists** use object syntax
  - An **ArrayList** expands as you add things to it
  - Arrays can hold primitives or objects, but **ArrayLists** can only hold objects

- Starting in Java 5, **ArrayLists** have been genericized
  - That means, every place you used to say **ArrayList**, you now have to say what kind of objects it holds; like this: **ArrayList<String>**
  - If you don’t do this, you will get a warning message, but your program will still run
Auto boxing and unboxing

• Earlier versions of Java wouldn’t let you use a primitive value where an object is required – you needed a “wrapper”
  – `ArrayList<Integer> myList = new ArrayList<Integer>();`
  – `myList.add(new Integer(5));`

• Since Java 1.5, this is automatic:
  – `myArrayList<Integer> myList = new myArrayList<Integer>();`
    `myList.add(5);`

• Similar statements exist for double/Double, boolean/Boolean, etc.
Generics

• A **generic** method is a method that is recompiled with different types as the need arises

• The bad news:
  – Instead of saying: `List words = new ArrayList();`
  – You'll have to say: `List<String> words = new ArrayList<String>();`

• The good news:
  – Replaces runtime type checks with compile-time checks
  – No casting; instead of `String title = (String) words.get(i);`
  – you use `String title = words.get(i);`

• Some classes and interfaces that have been “genericized” are: **Vector, ArrayList, LinkedList, Hashtable, HashMap, Stack, Queue, PriorityQueue, Dictionary, TreeMap** and **TreeSet**
Accessing with and without generics

• Using get the old way:
  – `ArrayList myList = new ArrayList();
    myList.add("Some string");
    String s = (String)myList.get(0);`

• Using get the new way:
  – `ArrayList<String> myList = new ArrayList<String>();
    myList.add("Some string");
    String s = myList.get(0);`

• Notice that casting is no longer necessary when we retrieve an element from a “genericized” `ArrayList`
private void printListofStrings(List<String> list) {
    for (int i = 0; i <= list.size(); i++) {
        System.out.println(list.get(i));
    }
}

• This method should be called with a parameter of type List<String>, but it can be called with a parameter of type List
  – The disadvantage is that the compiler won’t catch errors; instead, errors will cause a ClassCastException
  – This is necessary for backward compatibility
Writing your own generic types

• You have to tell the compiler that your class uses generic types using angle brackets, `<`
  – Syntax: `class Box<T>`
  – It is good style to use capital letters for the type
  – But if you have more than one generic type, you should use better names

• You cannot do much with generic types (obviously!), so if you need some functionality, then generics is not the way to go

• Eclipse example: Box.java
Summary

• If you think of a genericized type as a `type`, you won’t go far wrong
  – Use it wherever a type would be used
  – `ArrayList myList` becomes `ArrayList<String> myList`
  – `new ArrayList()` becomes `new ArrayList<String>()`
  – `public ArrayList reverse(ArrayList list)` becomes `public ArrayList<String> reverse(ArrayList<String> list)`

• Advantage: Instead of having collections of “Objects”, you can control the type of object

• Disadvantage: more complex, more typing