Generics
Arrays and ArrayLists

- 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

- To create an ArrayList:
  - ArrayList myList = new ArrayList();
  - Or, since an ArrayList is a kind of List,
    List myList = new ArrayList();

- To use an ArrayList,
  - boolean add(Object obj)
  - Object set(int index, Object obj)
  - Object get(int index)
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

- Java won’t let you use a primitive value where an object is required--you need a “wrapper”
  - `ArrayList<Integer> myList = new ArrayList<Integer>();`
  - `myList.add(new Integer(5));`
- Similarly, you can’t use an object where a primitive is required--you need to “unwrap” it
  - `int n = ((Integer)myArrayList.get(2)).intValue();`
- Java 1.5 makes this automatic:
  - `myArrayList<Integer> myList = new myArrayList<Integer>();`
    - `myList.add(5);`
    - `int n = myList.get(2);`
- Other extensions make this as transparent as possible
  - For example, control statements that previously required a `boolean` (if, while, do-while) can now take a `Boolean`
  - There are some subtle issues with equality tests, though
Generics

- A **generic** 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:
    ```java
    List<String> words = new ArrayList<String>();
    ```

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

- **Iterator** is an object that will let you step through the elements of a list one at a time
  - `List<String> listOfStrings = new ArrayList<String>();
    ...
    for (Iterator i = listOfStrings.iterator(); i.hasNext(); ) {
      String s = (String) i.next();
      System.out.println(s);
    }

- Iterators have also been genericized:
  - `List<String> listOfStrings = new ArrayList<String>();
    ...
    for (Iterator<String> i = listOfStrings.iterator(); i.hasNext(); ) {
      String s = i.next();
      System.out.println(s);
    }

- You can also use the new **for** statement (to be discussed)
private void printListOfStrings(List<String> list) {
    for (Iterator<String> i = list.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}

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
- Similarly, the Iterator need not be genericized as an Iterator<String>
Type wildcards

Here’s a simple (no generics) method to print out any list:

```java
private void printList(List list) {
    for (Iterator i = list.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}
```

The above still works in Java 1.5, but now it generates warning messages

- Java 1.5 incorporates lint (like C lint) to look for possible problems

You should eliminate all errors and warnings in your final code, so you need to tell Java that any type is acceptable:

```java
private void printListOfStrings(List<?> list) {
    for (Iterator<?> i = list.iterator(); i.hasNext(); ) {
        System.out.println(i.next());
    }
}
```
Writing your own generic types

- public class Box<T> {
  private List<T> contents;

  public Box() {
    contents = new ArrayList<T>();
  }

  public void add(T thing) { contents.add(thing); }

  public T grab() {
    if (contents.size() > 0) return contents.remove(0);
    else return null;
  }
}

- Sun’s recommendation is to use single capital letters (such as T) for types
- If you have more than a couple generic types, though, you should use better names
New **for** statement

- The syntax of the new statement is
  
  ```
  for(type var : array) {...}
  or  for(type var : collection) {...}
  ```

- Example:
  ```
  for(float x : myRealArray) {
    myRealSum += x;
  }
  ```

- For a collection class that has an Iterator, instead of
  ```
  for (Iterator iter = c.iterator(); iter.hasNext(); )
    ((TimerTask) iter.next()).cancel();
  ```
  you can now say
  ```
  for (TimerTask task : c)
    task.cancel();
  ```
The new `for` statement can also be used with arrays

Instead of

```java
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

you can say (assuming `array` is an `int` array):

```java
for (int value : array) {
    System.out.println(value);
}
```

Disadvantage: You don’t know the index of any of your values
Creating a ArrayList the old way

- The syntax for creating ArrayLists has changed between Java 1.4 and Java 5
- For compatibility reasons, the old way still works, but will give you warning messages
- Here are the (old) constructors:
  - `import java.util.ArrayList;`
  - `ArrayList vec1 = new ArrayList();`
    - Constructs an ArrayList with an initial capacity of 10
  - `ArrayList vec2 = new ArrayList(initialCapacity);`
Creating a **ArrayList** the new way

- Specify, in angle brackets after the name, the type of object that the class will hold

Examples:

- `ArrayList<String> vec1 = new ArrayList<String>();`
- `ArrayList<String> vec2 = new ArrayList<String>(10);`

To get the old behavior, but without the warning messages, use the `<?>` wildcard

- Example: `ArrayList<String> vec1 = new ArrayList<String>(1);`
Accessing with and without generics

- **Object get(int index)**
  - Returns the component at position `index`

- 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`
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