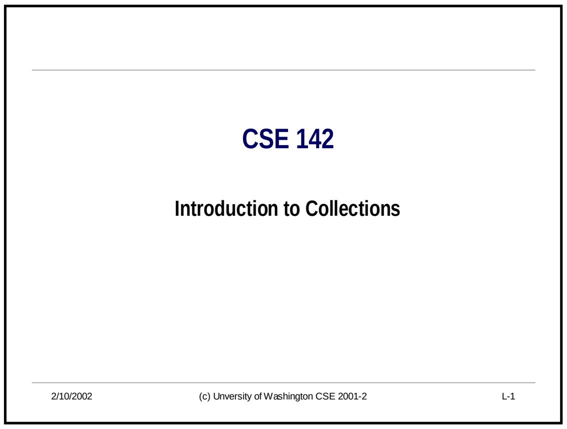
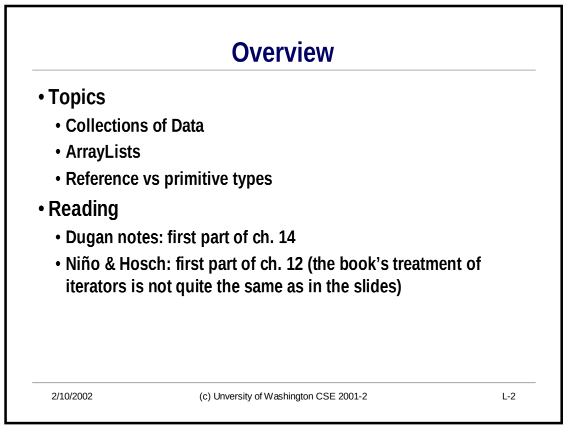


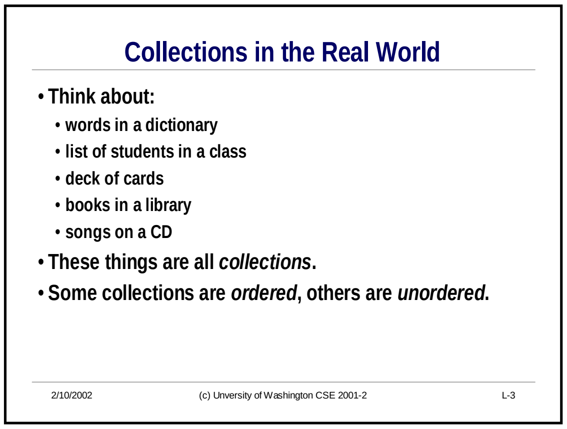
CSE 142

Introduction to Collections



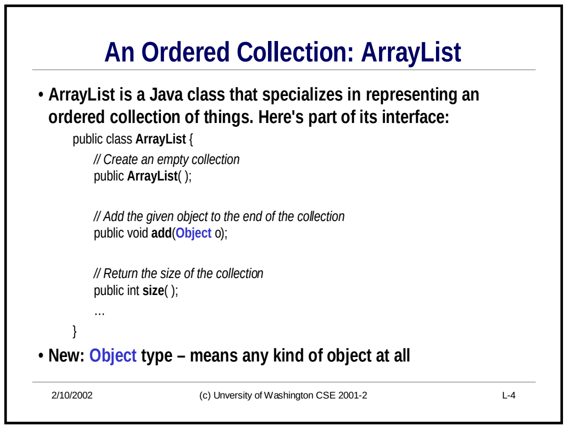
Overview

- Topics
  - Collections of Data
  - ArrayLists
  - Reference vs primitive types
- Reading
  - Dugan notes: first part of ch. 14
  - Niño & Hosch: first part of ch. 12 (the book's treatment of iterators is not quite the same as in the slides)



Collections in the Real World

- Think about:
  - words in a dictionary
  - list of students in a class
  - deck of cards
  - books in a library
  - songs on a CD
- These things are all *collections*.
- Some collections are *ordered*, others are *unordered*.



An Ordered Collection: ArrayList

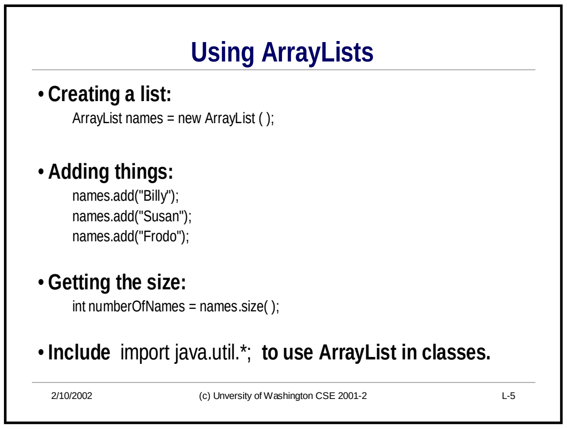
- ArrayList is a Java class that specializes in representing an ordered collection of things. Here's part of its interface:

```
public class ArrayList {
    // Create an empty collection
    public ArrayList( );

    // Add the given object to the end of the collection
    public void add(Object o);

    // Return the size of the collection
    public int size( );
    …
}
```

- New: Object type – means any kind of object at all



Using ArrayLists

- Creating a list:

```
ArrayList names = new ArrayList ( );
```

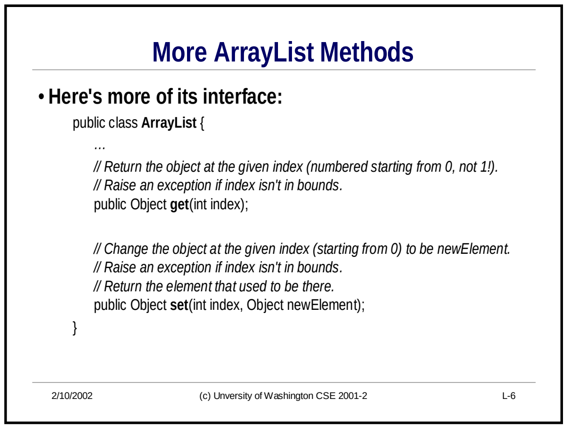
- Adding things:

```
names.add("Billy");
names.add("Susan");
names.add("Frodo");
```

- Getting the size:

```
int numberOfNames = names.size( );
```

- Include `import java.util.*;` to use ArrayList in classes.



More ArrayList Methods

- Here's more of its interface:

```
public class ArrayList {
    …
    // Return the object at the given index (numbered starting from 0, not 1!).
    // Raise an exception if index isn't in bounds.
    public Object get(int index);

    // Change the object at the given index (starting from 0) to be newElement.
    // Raise an exception if index isn't in bounds.
    // Return the element that used to be there.
    public Object set(int index, Object newElement);
}
```

More Using ArrayLists

- ArrayLists provide *indexed* access. We can ask for the *i*th item of the list, where the first item is at index 0, the second at index 1, and the last item is at index *n*-1 (where *n* is the size of the collection).

```
ArrayList names = new ArrayList ( );
names.add("Billy");
names.add("Susan");
```

- Java expressions:

```
names.get(0)
names.get(1)
```

A Problem

- Let's say we want to get things out of an ArrayList and name them.
- We might write the following:

```
public void printFirstName(ArrayList names) {
    String name = names.get(0);
    System.out.println("The first name is " + name);
}
```

- But BlueJ complains at the green line: "incompatible types: found: Object required: String“
  - (Jeva mutters something similar if you try this)
- Why? [Hint: look at the interface of the get method]

Object

- The return type of the method get() is Object.
- Think of Object as Java's way of saying "any type".
- All classes in Java (including the ones we write) have an "is-a" relationship to Object. In other words:
  - every String is an Object
  - every Rectangle is an Object
  - every ArrayList is an Object
- The reverse is not necessarily true!
  - every Object is not necessarily a String

Making False Claims

- We can say…

```
Object someObject = new Rectangle(. . .);
```

… because every Rectangle is an Object.

- In our example:

```
public void printFirstName(ArrayList names) {
    String name = names.get(0);
    System.out.println("The first name is " + name);
}
```

- We are claiming that an Object (the result of get) is a String, which is not necessarily true!
  - What if we passed an ArrayList of Rectangles to printFirstName?

Making Promises: Casting

- It looks like we're stuck. We can add things to the collection, but we can't get them back out!
- The solution is to make a promise.
  - Say we know that we've only placed String objects into the ArrayList. We can promise the compiler that the thing coming back out of the ArrayList is actually a String:

```
public void printFirstName(ArrayList names) {
    String name = (String)names.get(0);
    System.out.println("The first name is " + name);
}
```

- This promise is called a *cast*.

Casting (Review)

- Pattern:

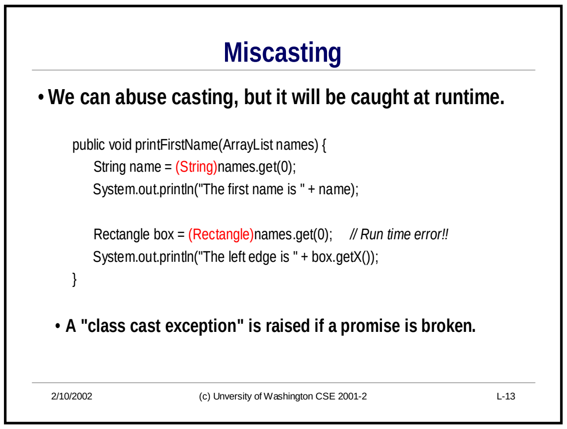
```
(<class-name>)<expression>
```

  - For example:

```
String name = (String)names.get(0);
```

- Casting does *not* change the type of the object. It is a promise that the object really is of the stated type.
- Casting also used for conversions, as we've seen.

```
(int) 3.1415927
```



Miscasting

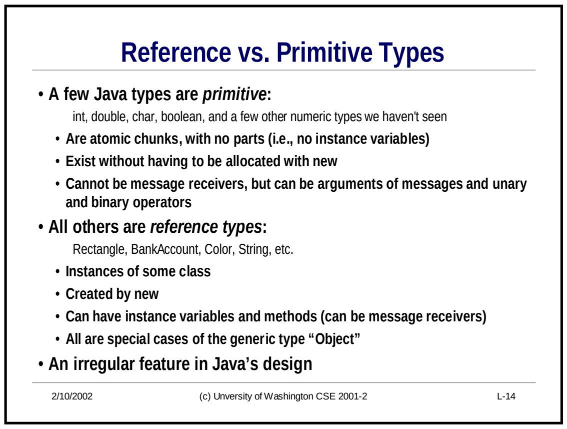
- We can abuse casting, but it will be caught at runtime.

```
public void printFirstName(ArrayList names) {
    String name = (String)names.get(0);
    System.out.println("The first name is " + name);

    Rectangle box = (Rectangle)names.get(0);     // Run time error!!
    System.out.println("The left edge is " + box.getX());
}
```

- A "class cast exception" is raised if a promise is broken.

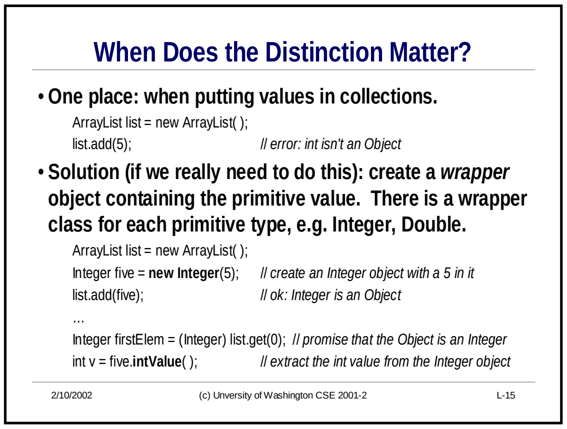
2/10/2002 (c) University of Washington CSE 2001-2 L-13



Reference vs. Primitive Types

- A few Java types are *primitive*:
  int, double, char, boolean, and a few other numeric types we haven't seen
  - Are atomic chunks, with no parts (i.e., no instance variables)
  - Exist without having to be allocated with new
  - Cannot be message receivers, but can be arguments of messages and unary and binary operators
- All others are *reference types*:
  Rectangle, BankAccount, Color, String, etc.
  - Instances of some class
  - Created by new
  - Can have instance variables and methods (can be message receivers)
  - All are special cases of the generic type "Object"
- An irregular feature in Java's design

2/10/2002 (c) University of Washington CSE 2001-2 L-14



When Does the Distinction Matter?

- One place: when putting values in collections.

```
ArrayList list = new ArrayList( );
list.add(5);                    // error: int isn't an Object
```

- Solution (if we really need to do this): create a *wrapper* object containing the primitive value. There is a wrapper class for each primitive type, e.g. Integer, Double.

```
ArrayList list = new ArrayList( );
Integer five = new Integer(5);    // create an Integer object with a 5 in it
list.add(five);                   // ok: Integer is an Object
...
Integer firstElem = (Integer) list.get(0);  // promise that the Object is an Integer
int v = five.intValue( );         // extract the int value from the Integer object
```

2/10/2002 (c) University of Washington CSE 2001-2 L-15