# Java Programming, Comprehensive
## Lecture 5

Bineet Sharma

# Agenda: Java Data Structures

- Arrays
- Collections
- Generics

# Java Data Structures: arrays

## Objectives

**Applied**

- Given a list of values or objects, write code that creates a one-dimensional array that stores those values or objects.
- Use for loops and enhanced for loops to work with the values or objects in an array.
- Use the methods of the Arrays class to fill an array, compare two arrays, sort an array, or search an array for a value.
- Implement the Comparable interface in any class you create.
- Create a reference to an array and copy elements from one array to another.
- Given a table of values or objects, write code that creates a two-dimensional array that stores those values or objects. The array can be either rectangular or jagged.
- Use for loops and enhanced for loops to work with the values or objects in a two-dimensional array.

# Java Data Structures: arrays

## Objectives (cont.)

**Applied (cont.)**

- Given the Java code for an application that uses any of the language elements presented in this chapter, explain what each statement in the application does.

**Knowledge**

- In general, explain what an array is and how you work with it.
- Describe the operation of the enhanced for loop and explain why it's especially useful with arrays.
- Explain when you need to implement the Comparable interface in a class you create.
- Explain what happens when you assign a new array to an existing array variable.
- Describe the difference between a rectangular array and a jagged array, and explain the difference in how you create them.

# Java Data Structures: arrays

- *array* is aggregate data type. Holds multiple values of same types of data
- An *element* is one of the items in an array
- Array is built-in in Java, for primitive or reference types
- Array allows you to manage large sets of data easily
- The syntax for declaring and instantiating an array:

**Two ways to declare an array**

```
type[] arrayName;
type arrayName[];      //allowed, but not preferred
```

**How to instantiate an array**

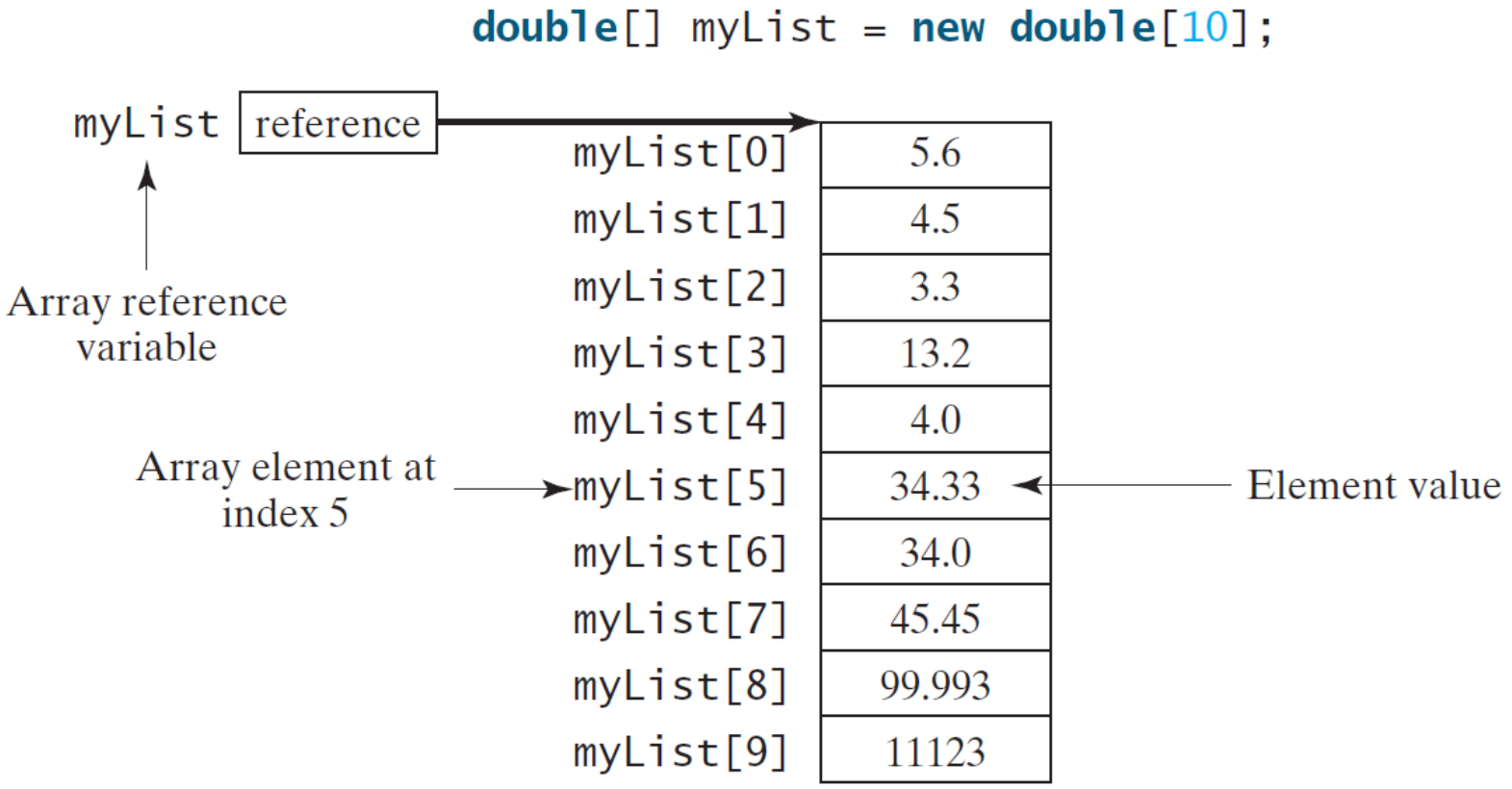
```
arrayName = new type[length];
```

**How to declare and instantiate an array in one statement**

```
type[] arrayName = new type[length];
```

# Java Data Structures: arrays

- Array is a data structure that represents a collection of the same types of data.



# Java Data Structures: arrays

**Code that declares an array of doubles**

```
double[] prices;
```

**Code that instantiates an array of doubles**

```
prices = new double[4];
```

**Code that declares and instantiates an array of doubles in one statement**

```
double[] prices = new double[4];
```

# Java Data Structures: arrays

## An array of String objects (really a references)

```
String[] titles = new String[3];
```

## An array of Product objects

```
Product[] products = new Product[5];
```

## Code that uses a constant to specify the array length

```
final int TITLE_COUNT = 100;  // size set at compile time
String[] titles = new String[TITLE_COUNT];
```

## Code that uses a variable to specify the array length

```
Scanner sc = new Scanner(System.in);
int titleCount = sc.nextInt();  // size set at runtime
String[] titles = new String[titleCount];
```

# Java Data Structures: arrays

## The syntax for referring to an element of an array

```
arrayName[index]    //0 based
```

## Code that assigns values to an array of double types

```
double[] prices = new double[4]; // 4 is size here
prices[0] = 14.95;
prices[1] = 12.95;
prices[2] = 11.95;
prices[3] = 9.95;
//prices[4] = 8.95;  // 4 is index, this would throw
                     // ArrayIndexOutOfBoundsException
```

## Code that assigns values to an array of String types

```
String[] names = new String[3];
names[0] = "Ted Lewis";
names[1] = "Sue Jones";
names[2] = "Ray Thomas";
```

# Java Data Structures: arrays

## Default values

```
When an array is created, its elements are assigned the
default value of:

0        = for all numeric primitive data types
'\u0000' = for char types
false    = boolean types
null     = reference types
```

# Java Data Structures: arrays

**Code that assigns objects to an array of Product objects**

```
Product[] products = new Product[2];
products[0] = new Product("java");
products[1] = new Product("jsps");
```

**The syntax for creating an array and assigning values in one statement**

```
type[] arrayName = {value1, value2, value3, ...};
```

**Examples that create an array and assign values in one statement**

```
double[] prices = {14.95, 12.95, 11.95, 9.95};
String[] names = {
    "Ted Lewis", "Sue Jones", "Ray Thomas"};
Product[] products = {
    new Product("java"), new Product("jsps")};
```

# Java Data Structures: arrays

## The syntax for getting the length of an array

```
arrayName.length
```

## Code that puts the numbers 0 through 9 in an array

```
int[] values = new int[10];
for (int i = 0; i < values.length; i++) {
   values[i] = i;
}
```

## Code that prints an array of prices to the console

```
double[] prices = {14.95, 12.95, 11.95, 9.95};
for (int i = 0; i < prices.length; i++) {
   System.out.println(prices[i]);
}
```

## The console output

```
14.95
12.95
11.95
9.95
```

# Java Data Structures: arrays

**Code that computes the average of the array of prices**

```
double sum = 0.0;
for (int i = 0; i < prices.length; i++)
{
    sum += prices[i];
}
double average = sum/prices.length;
```

**Another way to compute the average in a for loop**

```
double sum = 0.0;
for (int i = 0; i < prices.length; sum += prices[i++]);

double average = sum / prices.length;
```

# Java Data Structures: arrays

- Enhanced for loop

**The syntax of the enhanced for loop (JDK 1.5+)**

```
for (type variableName : arrayName) {
    statements
}
```

**Code that prints an array of prices to the console**

```
double[] prices = {14.95, 12.95, 11.95, 9.95};
for (double price : prices) {
   System.out.println(price);
}
```

**The console output**

```
14.95
12.95
11.95
9.95
```

- Will this loop suffice all of your *for* loop needs?

# Java Data Structures: Arrays

- Java API provides an *Arrays* class in java.util package
- Arrays' static methods is used for compare, sort, search, and copy arrays'.
- You can use Arrays *equals* method to check whether they contain the same number of elements with same values

**The Arrays class**

`java.util.Arrays`

**Static methods of the Arrays class**

- `fill(arrayName, value)`
- `fill(arrayName, index1, index2, value)`
- `equals(arrayName1, arrayName2)`
- `copyOf(arrayName, length)` `//shallow copy for ref type`
- `copyOfRange(arrayName, index1, index2)`
- `sort(arrayName)//ref type MUST implement Comparable`
- `sort(arrayName, index1, index2)`
- `binarySearch(arrayName, value)`

# Java Data Structures: Arrays

## Code that uses the fill method

```
int[] quantities = new int[5];
Arrays.fill(quantities, 1);  // all elements are set to 1
```

## Code that uses the fill method to fill 3 elements in an array

```
int[] quantities = new int[5];
Arrays.fill(quantities, 1, 4, 100); // elements 1, 2, and
                                    // 3 are set to 100
```

# Java Data Structures: Arrays

## Code that uses the equals method

```
String[] titles1 = {
    "War and Peace", "Gone With the Wind"};
String[] titles2 = {
    "War and Peace", "Gone With the Wind"};
if (titles1 == titles2)
    System.out.println("titles1 == titles2 is true");
else
    System.out.println("titles1 == titles2 is false");

if (Arrays.equals(titles1, titles2))
    System.out.println(
        "Arrays.equals(titles1, titles2) is true");
else
    System.out.println(
        "Arrays.equals(titles1, titles2) is false");
```

## The console output

```
titles1 == titles2 is false
Arrays.equals(titles1, titles2) is true
```

# Java Data Structures: Arrays

## Code that uses the sort method

```
int[] numbers = {2,6,4,1,8,5,9,3,7,0};
Arrays.sort(numbers);
for (int num : numbers)
{
    System.out.print(num + " ");
}
```

## The console output

```
0 1 2 3 4 5 6 7 8 9
```

## Code that uses the sort and binarySearch methods

```
String[] productCodes = {"mcbl", "jsps", "java"};
Arrays.sort(productCodes);
int index = Arrays.binarySearch(productCodes, "mcbl");
    // sets index to 2
```

# Java Data Structures: Arrays

## Will this work?

```
Product[] products = {
    new Product("jsps"), new Product("java")};
Arrays.sort(products);
int index = Arrays.binarySearch(products, "java");
    // will it set index to 0?
```

# Java Data Structures: array

## How about this?

```
Item[] items = new Item[3];
items[0] = new Item(102, "Duct Tape");
items[1] = new Item(103, "Bailing Wire");
items[2] = new Item(101, "Chewing Gum");
Arrays.sort(items);
for (Item i : items)
    System.out.println(i.getNumber() + ": " +
        i.getDescription());
```

## Will this give the output like this?

```
101: Chewing Gum
102: Duct Tape
103: Bailing Wire
```

# Java Data Structures: Arrays

- Making sort work for your class. E.g. Item class:

**The Comparable interface defined in the Java API**

```
public interface Comparable {
   int compareTo(Object obj);
}
```

**An Item class that implements the Comparable interface**

```
public class Item implements Comparable {
    private int number;
    private String description;

    public Item(int number, String description) {
        this.number = number;
        this.description = description;
    }

    public int getNumber() {
        return number;
    }
```

# Java Data Structures: Arrays

## An Item class that implements the Comparable interface (cont.)

```
    public String getDescription() {
        return description;
    }

    @Override
    public int compareTo(Object o) {
        Item i = (Item) o;  //risky?
        if (this.getNumber() < i.getNumber())
            return -1;
        if (this.getNumber() > i.getNumber())
            return 1;
        return 0;
    }
}
```

# Java Data Structures: array

**Now this code definitely sorts an array of Item objects the way we want**

```
Item[] items = new Item[3];
items[0] = new Item(102, "Duct Tape");
items[1] = new Item(103, "Bailing Wire");
items[2] = new Item(101, "Chewing Gum");
Arrays.sort(items);
for (Item i : items)
    System.out.println(i.getNumber() + ": " +
        i.getDescription());
```

**The console output**

```
101: Chewing Gum
102: Duct Tape
103: Bailing Wire
```

# Java Data Structures: array

- Can I sort by other data item? Yes, you can by writing another class which implements Comparator interface

**A LineItem class with multiple ways to sort**

```
public class LineItem implements Comparable{
    private Product product;
    private int quantity;

    public int compareTo(Object o) {
        if (o instanceof LineItem) {
            LineItem li = (LineItem) o;
            return product.getCode().compareTo
                  (li.product.getCode());
        }
        return 0;
    }

}
```

# Java Data Structures: array

## A LineItem class with multiple ways to sort (contd.)

- Add as many classes as you want the sorting done which implements Comparator interface

```
class LineItemQuantityCompare implements Comparator {

    public int compare(Object li1, Object li2) {
            int i1 = ((LineItem) li1).getQuantity();
            int i2 = ((LineItem) li2).getQuantity();
            if (i1 > i2) return 1;
            if (i1 < i2) return -1;
            return 0;
    }

    public boolean equals(Object li1, Object li2) {
            int i1 = ((LineItem) li1).getQuantity();
            int i2 = ((LineItem) li2).getQuantity();
            return (i1 == i2);
             return false;
    }
}
```

# Java Data Structures: array

## A LineItem class with multiple ways to sort (contd.)

- Now, create an array and sort in multiple different ways

```
LineItem[] items = new LineItem[3];

items[0] = new LineItem(
    new Product("dctp", "Duct Tape", 4.95), 10);
items[1] = new LineItem(
    new Product("blwr", "Bailing Wire", 14.95), 90);
items[2] = new LineItem(
    new Product("cgum", "Chewing Gum", 0.95), 15);

System.out.println("As they were inputted:");

for (LineItem i : items)
    System.out.println(i.getProduct().getCode()
                + ": " + i.getQuantity() + ": "
                + i.getProduct().getDescription()
    );
```

```
dctp: 10: Duct Tape
blwr: 90: Bailing Wire
cgum: 15: Chewing Gum
```

# Java Data Structures: Arrays

## A LineItem class with multiple way to sort (contd.)

- Now, create an array and sort in multiple different ways

```
Arrays.sort(items); //this will sort by code

for (LineItem i : items)
    System.out.println(i.getProduct().getCode()
                + ": " + i.getQuantity() + ": "
                + i.getProduct().getDescription()
    );
```

```
blwr: 90: Bailing Wire
cgum: 15: Chewing Gum
dctp: 10: Duct Tape
```

# Java Data Structures: Arrays

## A LineItem class with multiple way to sort (contd.)

- Now, create an array and sort in multiple different ways

```
Arrays.sort(items, new LineItemQuantityCompare());
//this will sort by quantity

for (LineItem i : items)
    System.out.println(i.getProduct().getCode()
                + ": " + i.getQuantity() + ": "
                + i.getProduct().getDescription()
    );
```

```
dctp: 10: Duct Tape
cgum: 15: Chewing Gum
blwr: 90: Bailing Wire
```

# Java Data Structures: array

## How to create a reference to an array

### Code that creates a reference to an array

```
double[] grades = {92.3, 88.0, 95.2, 90.5};
double[] percentages = grades;
percentages[1] = 70.2;            // changes grades[1] too
System.out.println("grades[1]=" + grades[1]);
                                  // prints 70.2
```

### Code that reuses an array variable

```
double[] grades = new double[5];
grades = new double[20]
```

# Java Data Structures: array

## How to copy an array with JDK 1.6 or later

### Code that copies the values of an array

```
double[] grades = {92.3, 88.0, 95.2, 90.5};
double[] percentages = Arrays.copyOf(
    grades, grades.length);
percentages[1] = 70.2;          // doesn't change grades[1]
System.out.println("grades[1]=" + grades[1]);
                                // prints 88.0
```

### Code that copies part of one array into another array

```
double[] grades = {92.3, 88.0, 95.2, 90.5};
Arrays.sort(grades);
double[] lowestGrades = Arrays.copyOfRange(grades, 0, 2);
double[] highestGrades = Arrays.copyOfRange(
                                    grades, 2, 4);
```

**Any use case?**

# Java Data Structures: array

## How to copy an array prior to JDK 1.6

### The syntax of the arraycopy method of the System class

```
System.arraycopy(
    fromArray, intFromIndex, toArray, intToIndex,
    intLength);
```

### Code that copies the values of an array

```
double[] grades = {92.3, 88.0, 95.2, 90.5};
double[] percentages = new double[grades.length];
System.arraycopy(
    grades, 0, percentages, 0, grades.length);
percentages[1] = 70.2;          // doesn't change grades[1]
System.out.println("grades[1]=" + grades[1]);
                                // prints 88.0
```

### Code that copies part of one array into another array

```
double[] grades = {92.3, 88.0, 95.2, 90.5};
Arrays.sort(grades);
double[] lowestGrades = new double[2];
System.arraycopy(grades, 0, lowestGrades, 0, 2);
double[] highestGrades = new double[2];
System.arraycopy(grades, 2, highestGrades, 0, 2);
```

# Java Data Structures: 2D array

- Two-Dimensional array: Array of one-dimensional array
- Implemented as array of arrays – called *rectangular array*
- Data is divided into row and column, with equal #of data in each row

**The syntax for creating a rectangular array**

```
type[][] arrayName = new type[rowCount][columnCount];
```

**A statement that creates a 3x2 array**

```
int[][] numbers = new int[3][2];
```

**The indexes for a 3x2 array**

```
[0][0]          [0][1]
[1][0]          [1][1]
[2][0]          [2][1]
```

# Java Data Structures: 2D array

## Code that assigns values to the array

```
numbers[0][0] = 1;
numbers[0][1] = 2;
numbers[1][0] = 3;
numbers[1][1] = 4;
numbers[2][0] = 5;
numbers[2][1] = 6;
```

## Code that creates a 3x2 array and initializes it in one statement

```
int[][] numbers = { {1,2}, {3,4}, {5,6} };
```

# Java Data Structures: 2D array

**Code that processes a rectangular array with nested for loops**

```
int[][] numbers = { {1,2}, {3,4}, {5,6} };
for (int i = 0; i < numbers.length; i++)
{
    for (int j = 0; j < numbers[i].length; j++)
        System.out.print(numbers[i][j] + "  ");
    System.out.print("\n");
}
```

**The console output**

```
1  2
3  4
5  6
```

# Java Data Structures: 2D *Jagged* array

- *Ragged/Jagged* array can hold uneven number of data elements in each row

```
0
1 2
3 4 5
```

**The syntax for creating a jagged array**

```
type[][] arrayName = new type[rowCount][];
```

**Code that creates a jagged array of integers**

```
int[][] numbers = new int[3][];
numbers[0] = new int[10];
numbers[1] = new int[15];
numbers[2] = new int[20];
```

**Code that creates and initializes a jagged array of strings**

```
String[][] titles =
  {{"War and Peace", "Wuthering Heights", "1984"},
   {"Casablanca", "Wizard of Oz", "Star Wars", "Birdy"},
   {"Blue Suede Shoes", "Yellow Submarine"}};
```

# Java Data Structures: 2D Jagged array

**Code that creates and initializes a jagged array of integers**

```
int number = 0;
int[][] pyramid = new int[3][];
for (int i = 0; i < pyramid.length; i++)
{
    pyramid[i] = new int[i+1];
    for (int j = 0; j < pyramid[i].length; j++)
        pyramid[i][j] = number++;
}
```

# Java Data Structures: 2D Jagged array

**Code that prints the contents of the jagged array of integers**

```
for (int i = 0; i < pyramid.length; i++)
{
   for (int j = 0; j < pyramid[i].length; j++)
      System.out.print(pyramid[i][j] + " ");
   System.out.print("\n");
}
```

**The console output**

```
0
1 2
3 4 5
```

**Using for-each (enhanced) loop to print a jagged arra**

```
for (int[] row : pyramid){
    for (int col : row)
        System.out.print(col + " ");
    System.out.print("\n");
}
```

# Java Data Structures: 2D Jagged array

**Code that prints the contents of the jagged array of integers**

```
for (int i = 0; i < pyramid.length; i++)
{
   for (int j = 0; j < pyramid[i].length; j++)
      System.out.print(pyramid[i][j] + " ");
   System.out.print("\n");
}
```

**The console output**

```
0
1 2
3 4 5
```

**What is the value of?**

```
pyramid.length    = ?
pyramid[0].length = ?
pyramid[1].length = ?
pyramid[2].length = ?
```

# Collections

# Java Data Structures: Collections

## Objectives

### Applied

- Given a list of values or objects, write code that creates an array list or linked list to store the values or objects. Then, write code that uses the values or objects in the list.
- Given a list of key-value pairs, write code that creates a hash map or tree map to store the entries. Then, write code that uses the entries in the list.
- Given Java code that uses any of the language elements presented in this chapter, explain what each statement does.

## Objectives (cont.)

### Knowledge

- Describe the similarities and differences between arrays and collections.
- Name the two main types of collections defined by the collection framework and explain how they differ.
- Describe the generics feature and explain how you use it to create typed collections and classes.
- Describe how the diamond operator works.
- Explain what an array list is and, in general, how it works.
- Explain what autoboxing is.
- Explain what a linked list is and, in general, how it works.
- Explain how you would decide whether to use an array list or a linked list for a given application.

# Java Data Structures: Collections

## Objectives (cont.)

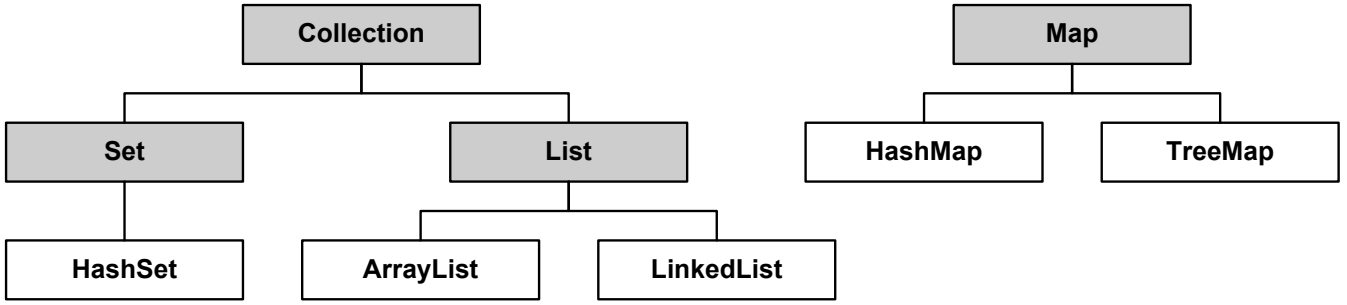
- Explain what a queue is and describe the two basic operations that a queue provides.
- Describe the main difference between a hash map and a tree map.
- Explain what the legacy collections are.
- Explain what an untyped collection is and what you must do to work with one.
- Explain when you need to use a wrapper class with untyped collections.

# Java Data Structures: Collections

- A collection is a container object that holds a group of objects (elements)
- The Java Collection Framework supports three types of collections:
  - Lists
  - Sets
  - Maps

# Java Data Structures: Collections

- The collection framework




**Collection interfaces**

- Collection //basic methods
- Set //no duplicate
- List //ordered
- Map //key value pair

**Common collection classes**

- ArrayList //almost like array (extendible, efficient)
- LinkedList //like array list, efficient in inserting
- HashSet //no duplicate
- HashMap //key is unique
- TreeMap //ordered automatically

# Java Data Structures: Collections

## How arrays and collections are similar

- Both can store multiple occurrences of objects.
- Some collection types (such as ArrayList) use arrays internally to store data.

## How arrays and collections differ

- An array is a Java language feature. Collections are classes in the Java API.
- Collection classes have methods that perform operations that arrays don't provide.
- Once created, arrays are fixed in size. Collections are variable in size.
- Arrays can store primitive types. Collections can't.
- Indexes are almost always required to process arrays. Collections are usually processed without using indexes.

# Java Data Structures: Collections

## Code that uses an array

```
String[] codes = new String[2];
codes[0] = "mcb2";
codes[1] = "java";
for (String s : codes)
    System.out.println(s);
```

## Code that uses a collection (old way)

```
ArrayList codes = new ArrayList();
codes.add("mcb2");
codes.add("java");
for (Object s : codes)
    System.out.println(s);
```

# Java Data Structures: Collections

## Code that stores Products in an untyped array list

```
// create an untyped array list
ArrayList products = new ArrayList();

// add three productss
products.add(new Product("dctp", "Duct Tape", 4.95));
products.add(new Product("blwr", "Bailing Wire", 14.95));
products.add(new Product("cgum", "Chewing Gum", 0.95));

// print the array list
for (int i = 0; i < products.size(); i++)
{
    Product p = (Product)products.get(i);
    System.out.println(p.getCode() + "\t" +
        p.getDescription() + "\t" +
        p.getFormattedPrice());
}
```

# Java Data Structures: Collections

## The compiler warning generated by the code

```
Type safety: The method add(Object) belongs to the
raw type ArrayList. References to generic type
ArrayList<E> should be parameterizedResulting
```

## Output

```
dctp    Duct Tape        $4.95
blwr    Bailing Wire     $14.95
cgum    Chewing Gum      $0.95
```

# Java Data Structures: Collections

## Wrapper classes for primitive types

| Primitive type | Wrapper class |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

# Java Data Structures: Collections

## Code that adds Integers to an untyped array list

```
ArrayList numbers = new ArrayList();

numbers.add(new Integer(1));
numbers.add(new Integer(2));
numbers.add(new Integer(3));
```

## Code that retrieves integers from the array list

```
for (int i = 0; i < numbers.size(); i++)
{
    int number = (Integer)numbers.get(i);
    System.out.println(number);
}
```

## Resulting output

```
1
2
3
```

# Java Data Structures: Collections

**Code that adds mixed data to an untyped array list**

```
ArrayList numbers = new ArrayList();

numbers.add(new Integer(1));
numbers.add(new Integer(2));
numbers.add("John");
numbers.add("Jack");

for (int i = 0; i < numbers.size(); i++)
{
    int number = (Integer)numbers.get(i);
    System.out.println(number);
}
```

**Resulting output**

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer
 at GenericQueue.main(GenericQueue.java:55)
```

- Where is the problem? What can you do?

# Java Data Structures: Collections

- Use collection in a generic array
- Compiler checks for compatibility

**Code that uses a collection (in a new generic way**

```
ArrayList<String> codes = new ArrayList<String>();
codes.add("mcb2");
codes.add("java");
codes.add("jsps");
codes.add(100); //compiler error

for (String s : codes)
    System.out.println(s);
```

# Java Data Structures: Collections

**The syntax for specifying the type of elements in a collection**

```
CollectionClass<Type> collectionName =
    new CollectionClass<Type>();
```

**A statement that creates an array list of type String**

```
ArrayList<String> codes = new ArrayList<String>();
```

**A statement that creates an array list of Integers**

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

**Code that creates a linked list of type Product**

```
LinkedList<Product> products;
products = new LinkedList<Product>();
```

# Java Data Structures: Collections

**The syntax for using type with JDK 1.7 or later**

```
CollectionClass<Type> collectionName =
    new CollectionClass<>(); //empty <> do here
```

**A statement that creates an array list of type String**

```
ArrayList<String> codes = new ArrayList<>();
```

# Java Data Structures: Collections

## The ArrayList class

`java.util.ArrayList`

## Constructors of the ArrayList class

- `ArrayList<E>()`
- `ArrayList<E>(intCapacity)`
- `ArrayList<E>(Collection)`

# Java Data Structures: Collections

## Common methods of the ArrayList class

- `add(object)`
- `add(index, object)`
- `clear()`
- `contains(object)`
- `get(index)`
- `indexOf(object)`
- `isEmpty()`
- `remove(index)`
- `remove(object)`
- `set(index, object)`
- `size()`
- `toArray()`

# Java Data Structures: Collections

## Code that uses an array list of type String

```
// create an array list of type String
ArrayList<String> codes = new ArrayList<>();
// add three strings
codes.add("mbdk");
codes.add("citr");
codes.add(0, "warp");
// print the array list
for (int i =0; i < codes.size(); i++)
{
    String code = codes.get(i);
    System.out.println(code);
}
```

## Resulting output

```
warp
mbdk
citr
```

# Java Data Structures: Collections

## Another way to display the contents of a collection

```
System.out.println(codes);
```

## Resulting output

```
[warp, mbdk, citr]
```

## Code that uses an array list of type Integer

```
ArrayList<Integer> numbers = new ArrayList<>();
//instead of:
//numbers.add(new Integer(1));
numbers.add(1); //Ok, due to autoboxing
numbers.add(2);
numbers.add(3);
System.out.println(numbers);
```

## Resulting output

```
[1, 2, 3]
```

# Java Data Structures: Collections

## Console output for the Invoice application

```
Welcome to the invoice application.

Enter product code: java
Enter quantity:     1
Another line item? (y/n): y

Enter product code: jsps
Enter quantity:     2
Another line item? (y/n): n

Code    Description                       Price   Qty     Total
----    -----------                       -----   ---     -----
java    Murach's Beginning Java           $49.50  1       $49.50
jsps    Murach's Java Servlets and JSP    $49.50  2       $99.00

                                          Invoice total:  $148.50
```

# Java Data Structures: Collections

## Classes used by the Invoice application

| Name | Description |
|---|---|
| Product | Represents a Product object. |
| ProductDB | Provides a getProduct method that retrieves the Product object for a specified product code. |
| Validator | Provides methods that accept and validate user input. |
| LineItem | Represents an invoice line item, which includes a Product object, a quantity, and a total. |
| Invoice | Represents a single invoice. The line items are represented by an array list. |
| InvoiceApp | Contains the main method for the Invoice application. |

# Java Data Structures: Collections

## The constructor for the Invoice class

- `Invoice()`

## The methods for the Invoice class

- `void addItem(LineItem lineItem)`
- `ArrayList getLineItems()`
- `double getInvoiceTotal()`
- `String getFormattedTotal()`

# Java Data Structures: Collections

## The code for the Invoice class

```
import java.text.NumberFormat;
import java.util.ArrayList;

public class Invoice
{
    // the instance variable
    private ArrayList<LineItem> lineItems;

    // the constructor
    public Invoice()
    {
        lineItems = new ArrayList<>();
    }

    // a method that adds a line item
    public void addItem(LineItem lineItem)
    {
        this.lineItems.add(lineItem);
    }
```

# Java Data Structures: Collections

## The code for the Invoice class (cont.)

```
    // the get accessor for the line item collection
    public ArrayList<LineItem> getLineItems()
    {
        return lineItems;
    }

    // a method that gets the invoice total
    public double getInvoiceTotal()
    {
        double invoiceTotal = 0;
        for (LineItem lineItem : this.lineItems)
        {
            invoiceTotal += lineItem.getTotal();
        }
        return invoiceTotal;
    }
    // a method that returns the invoice total
     public String getFormattedTotal(){
         NumberFormat currency =
             NumberFormat.getCurrencyInstance();
         return currency.format(this.getInvoiceTotal());
    }
}
```

# Java Data Structures: Collections

## The code for the InvoiceApp class

```
import java.util.Scanner;

public class InvoiceApp
{
    public static Invoice invoice = new Invoice();

    public static void main(String args[])
    {
        System.out.println(
            "Welcome to the invoice application.\n");
        getLineItems();
        displayInvoice();
    }

    public static void getLineItems()
    {
        Scanner sc = new Scanner(System.in);
        String choice = "y";
        while (choice.equalsIgnoreCase("y"))
        {
```

# Java Data Structures: Collections

## The code for the InvoiceApp class (cont.)

```
            // get the input from the user
            String productCode = Validator.getString(
                sc, "Enter product code: ");
            int quantity = Validator.getInt(
                sc, "Enter quantity:     ", 0, 1000);

            Product product =
                ProductDB.getProduct(productCode);
            invoice.addItem(
                new LineItem(product, quantity));

            // see if the user wants to continue
            choice = Validator.getString(
                sc, "Another line item? (y/n): ");
            System.out.println();
        }
    }
```

# Java Data Structures: Collections

## The code for the InvoiceApp class (cont.)

```
    public static void displayInvoice()
    {
        System.out.println(
            "Code\tDescription\t\t\tPrice\tQty\tTotal");
        System.out.println(
            "----\t-----------\t\t\t-----\t---\t-----");
        for (LineItem lineItem : invoice.getLineItems())
        {
            Product product = lineItem.getProduct();
            String s = product.getCode()
                + "\t" + product.getDescription()
                + "\t" + product.getFormattedPrice()
                + "\t" + lineItem.getQuantity()
                + "\t" + lineItem.getFormattedTotal();
            System.out.println(s);
        }
        System.out.println("\n\t\t\t\t\tInvoice total:\t"
            + invoice.getFormattedTotal() + "\n");
    }
}
```

# Java Data Structures: Collections

## The LinkedList class

`java.util.LinkedList`

## A constructor for the LinkedList class

- `LinkedList<E>()`

## Common methods of the LinkedList class

- `add(object)`
- `add(index, object)`
- `addFirst(object)`
- `addLast(object)`
- `clear()`
- `contains(object)`
- `get(index)`
- `getFirst()`
- `getLast()`

# Java Data Structures: Collections

## Common methods of the LinkedList class (cont.)

- `indexOf(object)`
- `peek()` //does not remove the first elem
- `offer(object)` //attempts to add at end
- `poll()` //removes first elem (null if empty)
- `remove()` //removes first elem(exception if empty)
- `remove(index)`
- `remove(object)`
- `removeFirst()`
- `removeLast()`
- `set(index, object)`
- `size()`
- `toArray()`

# Java Data Structures: Collections

## Code that creates a linked list of type String

```
// create a linked list of type String
LinkedList<String> codes = new LinkedList<>();

// add three strings
codes.add("mbdk");
codes.add("citr");
codes.add(0, "warp");

System.out.println(codes);
```

## Resulting output

```
[warp, mbdk, citr]
```

# Java Data Structures: Collections

**Code that adds elements to the beginning and end of the list**

```
codes.addFirst("wuth");
codes.addLast("wooz");

System.out.println(codes);
```

**Resulting output**

```
[wuth, warp, mbdk, citr, wooz]
```

**Code that uses an enhanced for loop to process the list :**

```
for (String s : codes)
    System.out.println(s);
```

**Resulting output**

```
wuth
warp
mbdk
citr
wooz
```

# Java Data Structures: Collections

## Code that retrieves the first and last elements of the list

```
String firstString = codes.removeFirst();
String lastString = codes.removeLast();
System.out.println(firstString);
System.out.println(lastString);
System.out.println(codes);
```

## Resulting output

```
wuth
wooz
[warp, mbdk, citr]
```

# Java Data Structures: Collections

## The HashMap and TreeMap classes

```
java.util.HashMap
java.util.TreeMap
```

## Common constructors of the HashMap and TreeMap classes

- `HashMap<K,V>()`
- `TreeMap<K,V>()`

# Java Data Structures: Collections

## Common methods of the HashMap and TreeMap classes

- `clear()`
- `containsKey(key)`
- `containsValue(value)`
- `entrySet()`
- `get(key)`
- `put(key, value)`
- `remove(key)`
- `size()`

## Common methods of the Map.Entry interface

- `getKey()`
- `getValue()`

# Java Data Structures: Collections

## Code that uses a hash map

```
// create an empty hash map
HashMap<String,String> books = new HashMap<>();

// add three entries
books.put("wooz", "Wizard of Oz");
books.put("mbdk", "Moby Dick");
books.put("wuth", "Wuthering Heights");

// print the entries
for (Map.Entry book : books.entrySet())
    System.out.println(book.getKey() + ": " +
        book.getValue());

// print the entry whose key is "mbdk"
System.out.println("\nCode mbdk is " +
    books.get("mbdk"));
```

# Java Data Structures: Collections

## Resulting output for code that uses a hash map

```
wuth: Wuthering Heights
mbdk: Moby Dick
wooz: Wizard of Oz

Code mbdk is Moby Dick
```

# Java Data Structures: Collections

## Code that uses a tree map

```
// create an empty tree map
TreeMap<String,String> books = new TreeMap<>();

// add three entries
books.put("wooz", "Wizard of Oz");
books.put("mbdk", "Moby Dick");
books.put("wuth", "Wuthering Heights");

// print the entries
for (Map.Entry book : books.entrySet())
    System.out.println(book.getKey() + ": " +
        book.getValue());

// print the entry whose key is "mbdk"
System.out.println("\nCode mbdk is " +
    books.get("mbdk"));
```

# Java Data Structures: Collections

## Resulting output for code that uses a tree map

```
mbdk: Moby Dick
wooz: Wizard of Oz
wuth: Wuthering Heights

Code mbdk is Moby Dick
```

# Java Data Structures: Collections

- Before Java Collections Framework was introduced in Java 2, several data structures were supported: Vector, Stack.
  - They are redesigned for Collections (old styles methods remains)

## Legacy collection classes

- Vector //similar to newer ArrayList,
- HashTable //similar to newer Hash Map
- Stack //implemented as stack. LinkedList is preferred

Old classes are synchronized – that is why they are slow, newer are preferred.

However, they are not deprecated and used where synchronization is important

# Java Data Structures: Collections

## Code that uses a vector

```
// create a vector
Vector codes = new Vector();

// add three strings
codes.add("mbdk");
codes.add("citr");
codes.add(0, "warp");

// print the vector
for (int i =0; i < codes.size(); i++)
{
    String code = (String)codes.get(i);
    System.out.println(code);
}
```

## Resulting output

```
warp
mbdk
citr
```

# Generics

# Java Data Structures: Generics

- Use the "**<>**" characters to designate the type to be used
  - wonder how ArrayList handles any types?
- We used them in Collections, because Collections uses generics (generalized type)
- Time to write our own Generics
- Generic allows generalized Types
- *Generics* abstracts over Types and provides readability and type safety during compile time
  - You can use generics with Methods, Classes and Interfaces as well

# Java Data Structures: Generic Methods

- Rationale:
  - Suppose you need to write a method to find out if an array contains a certain value (say Integer):

**Method that returns true if an Integer is found in the array**

```
public static boolean contains(Integer [] array,
                               Integer intObject) {
  for (Integer value : array) {
    if (intObject.equals(value))
         return true;
  }
return false; //did not find it
}
```

# Java Data Structures: Generic Methods

- Rationale (contd.):
  - Now, to test if a String array contains a String, you would write similar code

**Method that returns true if a String is found in the String array**

```
public static boolean contains(String [] array,
                               String strObject) {
    for (String value : array) {
       if (strObject.equals(value))
           return true;
    }
    return false; //did not find it
}
```

# Java Data Structures: Generic Methods

- Not so savvy, is it?
- Generic use of types solves the problem and we need only one method serving both the needs

**Method that returns true if a Generic Type <T> is found in the Generic Type <T> array**

```
public static <T> boolean contains(T[] array,
                                   T anyObject) {
   for (T value : array) {
      if (anyObject.equals(value))
            return true;
   }
   return false;
}
```

# Java Data Structures: Generic Methods

- Now you can use with any object

**You can use with Integer, or String**

```
Integer[] array = new Integer[5];

for (int j = 0; j < 5; j++) {
      array[j] =j * j;
}

if (contains(array, new Integer(16)) {
     System.out.println("Found the value");
}
else {
     System.out.println("Value not found");
}
```

# Java Data Structures: Generic Class

**Basic methods of a class that implements a generic queue – (you can create your own data structures like this)**

- `push(element)`
- `pull()`
- `size()`

# Java Data Structures: Generic Class

**The syntax for declaring a class that uses generic types**

```
public class ClassName<TypeVariable [,TypeVariable]...>{}
```

**A class statement for a class that implements a queue**

```
public class GenericQueue<E>{}
```

# Java Data Structures: Generic Class

**Code for a GenericQueue class for queue.**
**Generic allows parametric polymorphism**

```
import java.util.LinkedList;

public class GenericQueue<E> {
    private LinkedList<E> list = new LinkedList<>();

    public void push(E item)
    {
        list.addLast(item); //what is type of item?
    }

    public E pull()
    {
        return list.removeFirst();
    }

    public int size()
    {
        return list.size();
    }
}
```

# Java Data Structures: Generic Class

## Code that uses the GenericQueue class

```
GenericQueue<String> q1 = new GenericQueue<>();
q1.push("Item One");
q1.push("Item Two");
q1.push("Item Three");
System.out.println(
    "The queue contains " + q1.size() + " items");
while (q1.size() > 0)
    System.out.println(q1.pull());
System.out.println(
    "The queue now contains " + q1.size() + " items");
```

## Resulting output

```
The queue contains 3 items
Item One
Item Two
Item Three
The queue now contains 0 items
```

# Java Data Structures: Generic Class

**You can use this with any other objects**

```
GenericQueue<Integer> q2 = new GenericQueue<>();
q2.push(100);
q2.push(200);
q2.push(300);
q2.push(400);
System.out.println(
    "The queue contains " + q2.size() + " items");
while (q2.size() > 0)
    System.out.println(q2.pull());
System.out.println(
    "The queue now contains " + q2.size() + " items");
```

**Resulting output**

```
The queue contains 4 items
100
200
300
400
The queue now contains 0 items
```

- What changes you would need, to implement **LIFO**?

# Java Data Structures: Collections

## Console output for the enhanced Invoice application

```
Welcome to the invoice application.


Enter line items for invoice 1
Enter product code: java
Enter quantity:     1
Another line item? (y/n): n

Another invoice? (y/n): y


Enter line items for invoice 2
Enter product code: jsps
Enter quantity:     2
Another line item? (y/n): n

Another invoice? (y/n): n
```

# Java Data Structures: Collections

## Console output for the enhanced Invoice application (cont.)

```
You entered the following invoices:

Number  Total
------  -----
1       $49.50
2       $99.00
Total for all invoices: $148.50
```

# Java Data Structures: Collections

## Code for the InvoiceApp class

```
import java.util.Scanner;
import java.text.NumberFormat;

public class InvoiceApp
{
    private static GenericQueue<Invoice> invoices =
        new GenericQueue<>();

    private static Invoice invoice;

    private static Scanner sc;

    public static void main(String args[]) {
        System.out.println(
            "Welcome to the invoice application.\n");
        getInvoices();
        displayInvoices();
    }
```

# Java Data Structures: Collections

## Code for the InvoiceApp class (cont.)

```
public static void getInvoices() {
    sc = new Scanner(System.in);
    int invoiceNumber = 1;
    String anotherInvoice = "y";
    while (anotherInvoice.equalsIgnoreCase("y"))
    {
        invoice = new Invoice();
        System.out.println(
            "\nEnter line items for invoice "
            + invoiceNumber);
        getLineItems();
        invoices.push(invoice);

        // see if the user wants to continue
        anotherInvoice = Validator.getString(
            sc, "Another invoice? (y/n): ");
        System.out.println();
        invoiceNumber++;
    }
}
```

# Java Data Structures: Collections

## Code for the InvoiceApp class (cont.)

```
public static void getLineItems() {
    String anotherItem = "y";
    while (anotherItem.equalsIgnoreCase("y"))
    {
        // get the input from the user
        String productCode = Validator.getString(
            sc, "Enter product code: ");
        int quantity = Validator.getInt(
            sc, "Enter quantity:     ", 0, 1000);

        Product product =
            ProductDB.getProduct(productCode);
        invoice.addItem(
            new LineItem(product, quantity));

        // see if the user wants to continue
        anotherItem = Validator.getString(
            sc, "Another line item? (y/n): ");
        System.out.println();
    }
}
```

# Java Data Structures: Collections

## Code for the InvoiceApp class (cont.)

```
    public static void displayInvoices() {
        System.out.println(
            "You entered the following invoices:\n");
        System.out.println("Number\tTotal");
        System.out.println("------\t-----");
        double batchTotal = 0;
        int invoiceNumber = 1;
        while (invoices.size() > 0)
        {
            Invoice invoice = invoices.pull();
            System.out.println(invoiceNumber + "\t"
                + invoice.getFormattedTotal());
            invoiceNumber++;
            batchTotal += invoice.getInvoiceTotal();
        }
        NmmberFormat currency =
             NumberFormat.getCurrencyInstance();
        System.out.println(
            "Total for all invoices: "
            + currency.format(batchTotal));
    }
}
```

# Java Data Structures: Collections

- Which data structures to use?
  - Array: Bult-in type
    - Easy to use. Size is fixed. Random retrieve using index is fast. Insert/delete is slow
  - Collection: Different implementations
    - ArrayList: Easy as built in array (internally uses array). Size grows dynamically
    - LinkedList: Uses pointers to link to prev/next node in the list. Good in insert/delete. Not so in retrieving (goes sequentially)
    - TreeSet:
      - Ordered (natural or through Comparator)
      - log(n) performance for basic operations
    - HashSet:
      - Not sorted in order. Need a hashCode()
      - Constant time performance

# Further Reading

- More on collections/generics:
  - Generics are helpful tools for type safety.
  - Has complex syntax
  - You can generalize methods, and whole class
- More reading on Generics:
  - Subtyping, wildcard types, bounded wildcards
  - Erasure
  - http://docs.oracle.com/javase/tutorial/java/generics/index.html

# Next Lecture

- Working with data in:
  - Text files
  - Binary files
  - XML files

# Summary: Java Data Structures

- Arrays
- Collections
- Generics