Assignment 2

- Designing the music library system
- Due Tuesday, 8 pm
- No coding, just design
- You are free (and encouraged) to work with a partner
- Ask the Client

Review Class Design

Problem Description

- A TicketWizard Office needs a software system to track various events, their venues, and ticket orders for the events.
  - Each event has a name, description, date, time, a base ticket price and occurs at a single venue.
  - Each venue has a name, address, phone number.
  - Different events can have different seating plans. The seating plan consists of a number of sections and each section contains a number of seats. The price of a seat is determined by the base ticket price of the event and the section's price factor. A venue may host many different events, one event at a time, of course.

Problem Description (cont't)

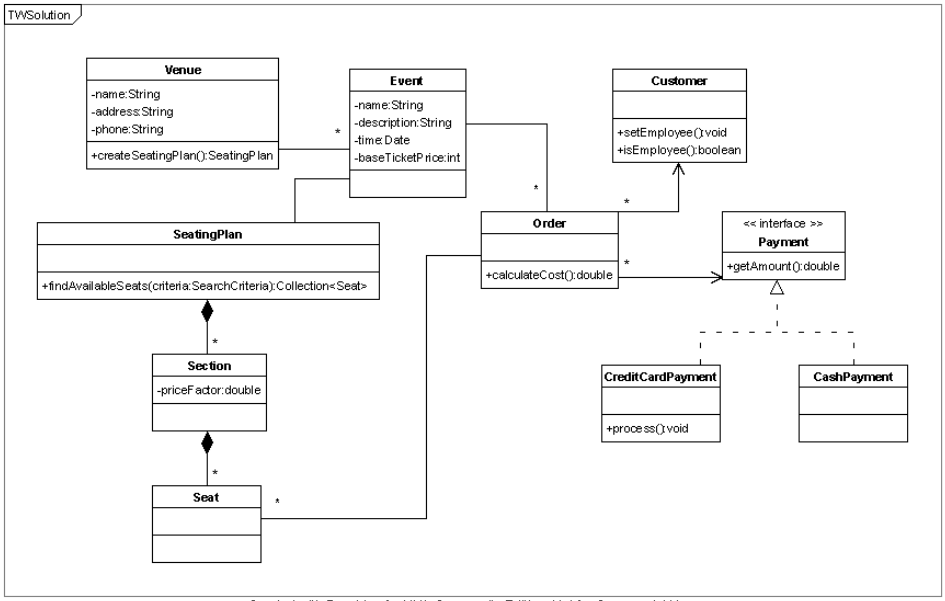
- Customers can place orders, which are made up of one or more seats for one or more events. Ticket office employees can also place orders; they enjoy a 10% discount on any regular ticket price.
- Customers can pay for their orders by cash or charge them to a credit card. For each order, the system must track the type of payment.
- Finally, the system must track customer information so that customers can be notified if the event is changed or cancelled.

Some Issues to consider

- Does a venue need to know about events? If so, how?
- Does an event need to know about venue? If so, how?
- Do we need Seat objects?
- Do we need Ticket objects?
- Do we need Customer objects?
- Do we need Employee objects?
- What other objects do we need?

How many errors can you find in this design?

Better Design



Key Concepts

- There are a lot of related concepts we covered
  - When you design a superclass, think about whether it might be extended in the future (i.e., which methods should be protected instead of private, etc.). This is the **open-closed principle** in action.
  - In Java, a subclass is considered a subtype as is an implementation of an interface. To ensure an instance of a subclass (or a class that extends an interface) is substitutable for its superclass (or its interface) we need to follow the **Liskov Substitutability Principle (LSP)**. i.e., watch out that pre-conditions and post-conditions of overridden methods do the right thing.
  - If we want to reuse code but can't do it via a subclass because we'd violate the LSP, we can use **delegation** where we keep an object of the type from which we want the code and we call the object's methods to do the work we want done.
  - If we want one class to act like different types, use **interfaces** (and sometimes delegation too!)

Introduction to Collections: the List interface

- compare and contrast the use of a List over an array
- know how and when to use a List data structure
- compare and contrast the use of generic data structures and arrays of type Object
- compare and contrast assignment with various generic collections under specific subclass scenarios
- use wildcards appropriately in generic type parameters to enable assignment in subclass scenarios

- **Reading**
  - 3rd & 4th Ed: Chapter 17 ; Skip 17.2
  - 2nd Ed: Chapter 22 ; Skip 22.2
- **Exercises**
  - 3rd & 4th Ed: Chapter 17, P17.1, P17.2, P17.3, P17.13
  - 2nd Ed: Chapter 22, P22.1, P22.2, P22.3, P22.13

Course Structure

- So far…
  - we've considered how to design and implement robust classes
- Now…
  - we're going to look at how to represent collections of information (objects) so that we can build programs that do more
- Then…
  - we're going to some programming concepts that will help you build even more interesting programs (Streams, GUI, Threads)

Why arrays aren't enough…

- Objects often have to store collections of references to other objects
  - e.g., a bank has a collection of accounts
- To this point, you have used arrays to store such collections

  e.g., `public class Bank {`
  `private Account[] accounts; …`
- But…
  - We have to decide the size of an array when we allocate it.
  - If the array fills, it doesn't expand automatically. We have to write code to create a bigger array and copy the data over.

Collections of Objects

- Sometimes we want to create objects that store a collection of other objects of an unspecified type.
- For example, we might want to create a list class that can store a list of any other type of object (e.g., a list of `String` or `Account` or `Point`).
- We can achieve this by storing the items in the collection in an array of type `Object`:

Collections of Objects: An Example

```
public class MyList {
   private Object[] items;
   private int numItems;
   final int INIT_NUM=10;

   public MyList {
      items = new Object[INIT_NUM];
      numItems = 0;
   }

   public void add( Object item ){…}
   public Object get( int index ){…}
   public boolean isEmpty(){…}
   public int size(){…}
}
```


Some Problems with MyList

- As a user of such a list, we have no problem adding items to the list:

```
myList.add( new Account() );
```

- …but we've got to be careful when we retrieve items from the list
  - we need to cast them to the appropriate type

```
Account acc = (Account) myList.get( 0 );
```

More Problems with MyList

- The fact that we can add *any* type of object to our `List` can be problematic.
- Suppose we want to create a list of `Account` objects:

```
MyList myList = new MyList();

myList.add( new Account() );
myList.add( new Account() );
myList.add( new Account() );
myList.add( new KitchenSink() );
myList.add( new Account() );
```

- The compiler won't flag the fact that we've added a `KitchenSink` to our list of `Account` objects – ugh.

List

- A `List` is an interface defined in the Java libraries.
- An object of type `List` acts like an array except that it automatically grows and shrinks as needed.
- There are several kinds of `List` classes which differ in their performance characteristics
  - `ArrayList, Vector, LinkedList`, etc..
  - Details are described in CPSC 221
  - We will use an `ArrayList` for this lecture

List

- A `List` is an example of a ***generic interface/class***.
- We specify the type of data to be stored in the list when a `List` is declared and instantiated:
  - `List<Account> accts = new ArrayList<Account>();`
    `// a list of Account objects`
  - `List<String> strings = new ArrayList<String>();`
    `// a list of String objects`

List

- The compiler will not allow us to add objects of the wrong type:
  - `List<Account> accts = new ArrayList<Account>();`

```
accts.add( new Account() );      // OK
accts.add( new Account() );      // OK
accts.add( new KitchenSink() );   // won't compile
```

- This is a good thing. The compiler will now check that we're adding the right type of object to our list.

List

- It's also easy to retrieve items from the list.
- Recall that when we retrieve an item from `MyList`, we have to cast to the appropriate type.

  The cast is not necessary when working with a generic `List`.
- Let's assume that we're working with the `List` declared on the previous page and that we've inserted a few `Account` objects into the list:

```
Account myAccount = accts.get( 0 );
// Gets the account at position 0 in the list
```

- No cast is necessary.

List

- Given:

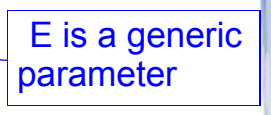
```
List<Account> accts = new ArrayList<Account>();
```

  - we can add objects of type `Account`
  - we can also add objects that are a subtype of `Account`
- So, if `SavingsAccount` is a subclass of `Account`, we can do the following:

```
accts.add( new Account() );
accts.add( new SavingsAccount() );
```

List Methods

- `List` has many useful methods:
  public interface List<E> {




```
...
public boolean add( E item )
   // add at end of list
public boolean add( int i, E item )
   // insert at specific position i
public boolean contains( Object item )
   // is item in the accounts collection
public E get( int i )
   // get item at position i
public E remove( int i )
   // remove account at position i
public int size()
   // gets number of elements in list
   // NOT current capacity of list
...
```


Java Generics

- Note that the `E` in the `List` API is a ***generic parameter*** (or *type parameter)*
- `E` represents the **type** that is specified by the client when the `List` is declared and instantiated
- For example:

```
List<Account> accList;
// E is Account

List<String> strList;
// E is String
```

- For the API for this interface, see the online documentation:
  http://java.sun.com/javase/6/docs/api/index.html

Generic Programming

- *Generic programming* is the creation of programming constructs that can be used with many different types
- A *generic class* has one or more type variables, e.g.
  - public class ArrayList<E>
- These type variables can be instantiated with class or interface types

Type Variables

- The type that you supply replaces the type variable in the class or interface, e.g.
  - ArrayList<Account>
- Type variables make generic code safer and easier to read
- By the way, E means “element type in a collection.”

Good Type Variable Names

Type Variable	Name Meaning
E	Element type in a collection
K	Key type in a map
V	Value type in a map
T	General type
S, U	Additional general types

Type Variables

- You cannot use primitive types as type parameters, e.g.
  - There is no ArrayList<int> or ArrayList<double>
- Use the Wrapper class instead, e.g.
  - Integer and Double

Instantiating a Generic Class

GenericClassName<Type1, Type2, . . .>

Example:

```
ArrayList<BankAccount>
HashMap<String, Integer>
```

Purpose:
To supply specific types for the type variables of a generic class.

Example using List

```
public class Bank {
    private List<Account> accounts;

    public Bank() {
        accounts = new ArrayList<Account>();
    }

    // Add new account at the end of List
    public void newAccount(double balance) {
        accounts.add(new Account(balance));
    }

    // Get number of accounts at Bank
    public int getNumAccounts() {
        return accounts.size();
    }

…
}
```

Another Generic Example

```
public class Pair<T, S>
{
   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }

   private T first;
   private S second;
}
```


Another Generic Example

How would you use the generic `Pair` class to construct a pair of strings `"Hello"` and `"World"`?

Answer: `new Pair<String, String>("Hello", "World")`

In-Class Exercise I

- Complete the following method that counts the number of times a particular string is found in an `List`

```
public static int count(List<String> list,
                                  String toFind )
{
```


Tea break!

Generic Methods

- *Generic method*: method with a type variable
- Can be defined inside ordinary and generic classes
- A regular (non-generic) method:

```
/**
   Prints all elements in an array of strings.
   @param a the array to print
*/
public static void print(String[] a)
{
   for (String e : a)
      System.out.print(e + " ");
      System.out.println();
}
```


Continued

Generic Methods (cont.)

- What if we want to print an array of Rectangle objects instead?

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Generic Methods

- When calling a generic method, you need not instantiate the type variables:

```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

- The compiler deduces that `E` is `Rectangle`
- You can also define generic methods that are not static
- You can even have generic methods in generic classes

Defining a Generic Method

```
modifiers <TypeVariable1, TypeVariable2, . . .> returnType
methodName(parameters)
{
   body
}
```

Example:

```
public static <E> void print( E[] a)
{
   . . .
}
```

Purpose:

To define a generic method that depends on type variables.

Generic Methods

Is the `getFirst` method of the `Pair` class a generic method?

Answer: No – the method has no type parameters. It is an ordinary method in a generic class.

Assignment with Arrays and subclasses

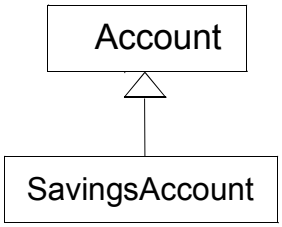
- Assume that `SavingsAccount` is a subclass of `Account`. Consider this:

```
Account[] acc = new Account[10];
SavingsAccount[] sAcc = new SavingsAccount[10];
```

- Is this allowed?

```
acc[0] = new SavingsAccount();

SavingsAccount sa = acc[0];
```



Assignment with Arrays and subclasses

- What about this?

```
acc = sAcc;
```

This *does* compile but can lead to problems if we then do

```
acc[0] = new Account();

// oops - just put an Account into an array
// of SavingsAccount objects
:
```

…problem isn't detected by compiler.

An exception is thrown when the program runs – nasty.

Assignment with Generics

- Consider this:

```
List<Account> accounts
          = new ArrayList<Account>();

List<SavingsAccount> savingsAccounts
          = new ArrayList<SavingsAccount>();
```

- Is this allowed?

```
accounts.add( 0, new SavingsAccount() );


SavingsAccount sa = accounts.get( 0 );
```

Assignment with Generics

- What about this?

```
accounts = savingsAccounts;
```

- This code does *not* compile and so the problem illustrated with arrays earlier is avoided.

Assignment with Generics

- Assume we have the method:

```
public void myMethod(List<Account> list ) {…}
```

then the following client call will also not compile:

```
List<SavingsAccount> savAccs
       = new ArrayList<SavingsAccount>();

myMethod( savAccs );
```

Assignment with Generics

- Note that List<SavingsAccount> is not a subclass of List<Account>
  - Even though SavingsAccount is a subclass of Account
- Inheritance of type parameters does not lead to inheritance of generic classes
- This restriction saves us some trouble, as just shown

Arrays and ArrayLists

- Let's look at another example
  - from Head First Java

A simple Animal class hierarchy

```
abstract class Animal {
    void eat() {
        System.out.println("animal eating");
    }
}
public class Dog extends Animal {
void bark() { }
}
public class Cat extends Animal {
void meow() {}
}
```


Arrays

- Let's consider arrays first
- Let's create an array of Animals that hold both cats and dogs
- Let's also create an array of Dogs that can hold only dogs

Arrays

```
public class TestGenerics1 {

public static void main(String[]
args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new
Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(),
new Dog()};

takeAnimals(animals);

takeAnimals(dogs);

}
```

```
public void takeAnimals(Animal[]
animals)

{

for(Animal a: animals)

{

a.eat();

}

}


}
```


Arrays

```
public class TestGenerics1 {

public static void main(String[]
args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new
Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(),
new Dog()};

takeAnimals(animals);

takeAnimals(dogs);

}
```

```
public void takeAnimals(Animal[]
animals)

{

for(Animal a: animals)

{

a.eat();

}

}
```

Create Animal array

Create Dog array

Call takeAnimals() on each of them

Arrays

```
public class TestGenerics1 {

public static void main(String[]
args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new
Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(),
new Dog()};

takeAnimals(animals);

takeAnimals(dogs);
}
```

```
public void takeAnimals(Animal[]
animals)

{

for(Animal a: animals)

{

a.eat();

}

}


}
```

We can call ONLY the methods declared in type Animal since the parameter is an Animals array

Arrays

```
public class TestGenerics1 {

public static void main(String[]
args) {

new TestGenerics1().go();

}


public void go(){

Animal[] animals = {new Dog(), new
Cat(), new Dog()};

Dog[] dogs = {new Dog(), new Dog(),
new Dog()};

takeAnimals(animals);

takeAnimals(dogs);
}
```

```
public void takeAnimals(Animal[]
animals)

{

for(Animal a: animals)

{

a.eat();

}

}


}
```

>
animal eating
animal eating
animal eating
animal eating
animal eating
animal eating

ArrayLists

- That was using Arrays
- Let's try the same thing with ArrayLists

ArrayLists

```
import java.util.*;

public class TestGenerics2 {

public static void main(String[]
args) {

new TestGenerics2().go();

}


public void go(){

    ArrayList<Animal> animals = new
ArrayList<Animal>();

    animals.add(new Dog());

    animals.add(new Cat());

    animals.add(new Dog());

takeAnimals(animals);

}
```

```
public void
takeAnimals(ArrayList<Animal>
animals)

{

for(Animal a: animals)

{

a.eat();


}

}
```

We've just changed from Animal[] to ArrayList<Animal>
We create an ArrayList of Animals containing Cats and Dogs, and call the takeAnimals() method

ArrayLists

```
import java.util.*;

public class TestGenerics2 {

public static void main(String[]
args) {

new TestGenerics2().go();

}


public void go(){

    ArrayList<Animal> animals = new
ArrayList<Animal>();

    animals.add(new Dog());

    animals.add(new Cat());

    animals.add(new Dog());
 takeAnimals(animals);

}
```

```
public void
takeAnimals(ArrayList<Animal>
animals)

{

for(Animal a: animals)

{

a.eat();


}
}
}
```

The method takes an ArrayList<Animal>.
The output is:
>
animal eating
animal eating
animal eating

ArrayLists

- So far, so good
- With the Array example, we were able to pass a Dog array to a method that took an Animal array parameter
- What happens if we pass an ArrayList<Dog> to our takeAnimals() method, which takes ArrayList<Animal> as a parameter?

ArrayLists

```
public void go(){
    ArrayList<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Dog());
    dogs.add(new Dog());
    takeAnimals(dogs);
}
public void takeAnimals(ArrayList<Animal> animals){
       for(Animal a: animals)
       {
           a.eat();
       }
    }
```


ArrayLists

```
public void go(){
    ArrayList<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Dog());
    dogs.add(new Dog());
    takeAnimals(dogs);
}
public void takeAnimals(ArrayList<Animal> animals){
       for(Animal a: animals)
       {
           a.eat();
       }
    }
```

```
Exception in thread "main"
java.lang.Error: Unresolved compilation
problem:
      The method
takeAnimals(ArrayList<Animal>) in the
type TestGenerics2 is not applicable for
the arguments (ArrayList<Dog>)

      at
TestGenerics2.go(TestGenerics2.java:13)
      at
TestGenerics2.main(TestGenerics2.java:5
)
```

Arrays, ArrayLists, and Polymorphism

- With arrays, we could pass a Dog array to a method expecting an Animal array
  - Polymorphism in action
  - Dog IS-A Animal
- We lost this ability with ArrayLists
- What if we *were* allowed to pass an ArrayList<Dog> to that method? What would happen?
  - Just hypothetically (Java won't let us)

ArrayLists

- What's the worst that could happen?

```
public void takeAnimals(ArrayList<Animal> animals){
animals.add(new Cat()); // bad! A Cat in what should
                        // have been a Dogs-only
                        // ArrayList
```

- So Java just won't let you take this risk
- If you declare a method to take ArrayList<Animal> it can take ONLY an ArrayList<Animal>, not ArrayList<Dog> or ArrayList<Cat>

Arrays and ArrayLists

- So why could we do that with Arrays but not ArrayLists?
  - We could pass a Dog array to a method that takes an Animal array
  - Couldn't somebody add a Cat to the Dog array?
  - Yes! And unfortunately it *would* compile and the error wouldn't be caught until runtime

Runtime

```
takeAnimals(dogs);


public void takeAnimals(Animal[] animals)
{
animals[0] = new Cat();
for(Animal a: animals)
{
a.eat();
}
}
```

```
Exception in thread "main"
java.lang.ArrayStoreException: Cat
        at
TestGenerics1.takeAnimals(TestGenerics1.java:1
9)
        at TestGenerics1.go(TestGenerics1.java:14)
        at TestGenerics1.main(TestGenerics1.java:6)
```

ArrayList

- With ArrayLists, we avoid this nasty problem because type checking occurs when we compile

Motivating Wildcards

- Imagine that we want to add a method to `Bank` that will take a list of accounts and send a directed advertisement to their owners

```
public void spam(List<Account> targetAccounts) ...
```

- We have a problem. We may want to spam a list of `SavingsAccount` but we cannot write:

```
List<SavingsAccount> savingsAccounts
       = new ArrayList<SavingsAccount>();
Bank b = new Bank();

b.spam( savingsAccounts );   //not allowed
```

Bounded Wildcards

- In such cases we can use wildcards in the type parameter:

```
public void spam(
  List<? extends Account> targetAccounts )
{…}
```

- `<? extends Account>` indicates that we can pass a `List` of any type that is a subtype of `Account`
- So we can now pass a `List` of `Account` or `SavingsAccount` or any other type that's a subtype of `Account`.

Bounded Wildcards - Question

- When we use a bounded wildcard, we can visit the items in the collection but we are not allowed to add an item to the collection.

```
public void spam(List<? extends Account>
targetAccounts )
{
   targetAccounts.add( new Account() );
   //…
}
```

- Why is this not allowed?

Bounded Wildcards - Question

- We can answer that by revisiting our Animals/Dogs/Cats example
- We discovered that we could not pass ArrayList<Dog> to a method expecting an ArrayList<Animal> parameter
- But now we know about a workaround: bounded wildcards

Bounded Wildcards

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
  for (Animal a : animals){
    a.eat();
    }
}
```


Now we can pass in an ArrayList<Dog> or ArrayList<Cat>

Bounded Wildcards

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
  for (Animal a : animals){
    a.eat();
    }
}
```

But what's the difference? Don't we have the same problem as before? This allows us to pass in an ArrayList<Dog> but somebody could still add a Cat to the ArrayList of Dogs, right?

Bounded Wildcards

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
  for (Animal a : animals){
    a.eat();
    }
}
```

But what's the difference? Don't we have the same problem as before? This allows us to pass in an ArrayList<Dog> but somebody could still add a Cat to the ArrayList of Dogs, right?

No! When you use a bounded wildcard in a method parameter, the compiler will not let you add anything to that list. You can use the list but not add anything to it. Problem solved.

Wildcard Types

Name	Syntax	Meaning
Wildcard with lower bound	? extends B	Any subtype of B
Wildcard with upper bound	? super B	Any supertype of B
Unbounded wildcard	?	Any type

Constraining Type Variables

- Very occasionally, you need to supply two or more type bounds

```
<E extends Comparable & Cloneable>
```

- `extends`, when applied to type variables, actually means "extends or implements"
- The bounds can be either classes or interfaces
- Type variable can be replaced with a class or interface type

Using ArrayLists

- We started by introducing the List interface and ArrayList implementation, and took a bit of a detour through generic programming
- Let's look at how to use ArrayLists in more detail

Array Lists

- The `ArrayList` class manages a sequence of objects
- Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- The `ArrayList` class is a generic class: `ArrayList<T>` collects objects of type `T`:

```
ArrayList<BankAccount> accounts = new
       ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

- `size` method yields number of elements

Retrieving Array List Elements

- Use `get` method
- Index starts at 0
-
```
BankAccount anAccount = accounts.get(2); // gets the
third element of the array list
```
- Bounds error if index is out of range
- Most common bounds error:

```
int i = accounts.size();
anAccount = accounts.get(i); // Error
//legal index values are 0. . .i-1
```


Adding Elements

- `set` overwrites an existing value

```
BankAccount anAccount = new BankAccount(1729);
accounts.set(2, anAccount);
```

- `add` adds a new value before the index

```
accounts.add(i, a)
```

Continued

Adding Elements (cont.)

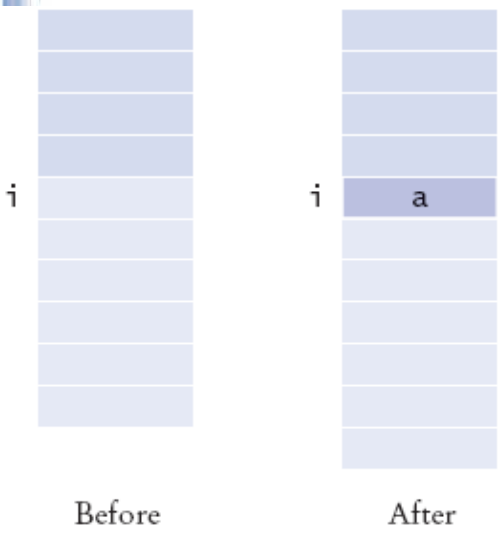


Figure 3 Adding an Element in the Middle of an Array List

Removing Elements

`remove` removes an element at an index

```
accounts.remove(i)
```

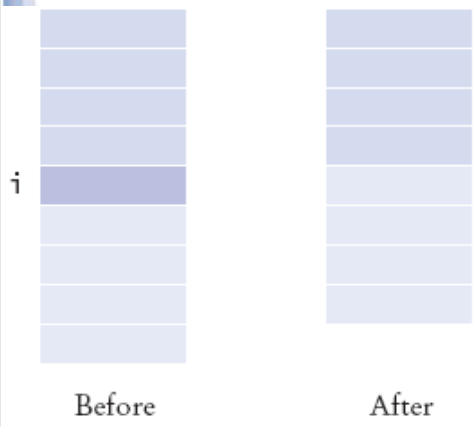


Figure 4 Removing an Element from the Middle of an Array List

ch07/arraylist/ArrayListTester.java

```
01: import java.util.ArrayList;
02:
03: /**
04:    This program tests the ArrayList class.
05: */
06: public class ArrayListTester
07: {
08:    public static void main(String[] args)
09:    {
10:       ArrayList<BankAccount> accounts
11:             = new ArrayList<BankAccount>();
12:       accounts.add(new BankAccount(1001));
13:       accounts.add(new BankAccount(1015));
14:       accounts.add(new BankAccount(1729));
15:       accounts.add(1, new BankAccount(1008));
16:       accounts.remove(0);
17:
18:       System.out.println("Size: " + accounts.size());
19:       System.out.println("Expected: 3");
20:       BankAccount first = accounts.get(0);
```

Continued 77

ch07/arraylist/ArrayListTester.java (cont.)

```
21:       System.out.println("First account number: "
22:             + first.getAccountNumber());
23:       System.out.println("Expected: 1015");
24:       BankAccount last = accounts.get(accounts.size() - 1);
25:       System.out.println("Last account number: "
26:             + last.getAccountNumber());
27:       System.out.println("Expected: 1729");
28:    }
29: }
```

ch07/arraylist/BankAccount.java

```
01: /**
02:    A bank account has a balance that can be changed by
03:    deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:    /**
08:       Constructs a bank account with a zero balance
09:       @param anAccountNumber the account number for this account
10:    */
11:    public BankAccount(int anAccountNumber)
12:    {
13:       accountNumber = anAccountNumber;
14:       balance = 0;
15:    }
16:
17:    /**
18:       Constructs a bank account with a given balance
19:       @param anAccountNumber the account number for this account
20:       @param initialBalance the initial balance
21:    */
```

Continued 79

ch07/arraylist/BankAccount.java (cont.)

```
22:    public BankAccount(int anAccountNumber, double initialBalance)
23:    {
24:       accountNumber = anAccountNumber;
25:       balance = initialBalance;
26:    }
27:
28:    /**
29:       Gets the account number of this bank account.
30:       @return the account number
31:    */
32:    public int getAccountNumber()
33:    {
34:       return accountNumber;
35:    }
36:
37:    /**
38:       Deposits money into the bank account.
39:       @param amount the amount to deposit
40:    */
41:    public void deposit(double amount)
42:    {
43:       double newBalance = balance + amount;
44:       balance = newBalance;
45:    }
```

Continued 80

ch07/arraylist/BankAccount.java (cont.)

```
46:
47:    /**
48:       Withdraws money from the bank account.
49:       @param amount the amount to withdraw
50:    */
51:    public void withdraw(double amount)
52:    {
53:       double newBalance = balance - amount;
54:       balance = newBalance;
55:    }
56:
57:    /**
58:       Gets the current balance of the bank account.
59:       @return the current balance
60:    */
61:    public double getBalance()
62:    {
63:       return balance;
64:    }
65:
66:    private int accountNumber;
67:    private double balance;
68: }
```

Continued

ch07/arraylist/BankAccount.java (cont.)

Output:

```
Size: 3
Expected: 3
First account number: 1008
Expected: 1008
Last account number: 1729
Expected: 1729
```

Arrays and ArrayLists

How do you construct an array of 10 strings? An array list of strings?

Answer:

```
new String[10];
new ArrayList<String>();
```


ArrayLists

What is the content of `names` after the following statements?

```
ArrayList<String> names = new ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

Answer: `names` contains the strings `"B"` and `"C"` at positions 0 and 1

Wrappers

- You cannot insert primitive types directly into array lists
- To treat primitive type values as objects, you must use wrapper classes:

```
ArrayList<Double> data = new ArrayList<Double>();
data.add(29.95);
double x = data.get(0);
```

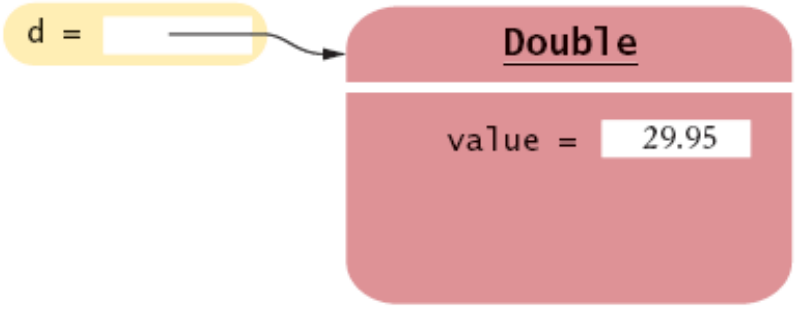


Figure 5 An Object of a Wrapper Class

Wrappers

There are wrapper classes for all eight primitive types:

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Auto-boxing

- Auto-boxing: Starting with Java 5.0, conversion between primitive types and the corresponding wrapper classes is automatic.

```
Double d = 29.95; // auto-boxing; same as Double d =
          new Double(29.95);
double x = d; // auto-unboxing; same as double x =
          d.doubleValue();
```

- Auto-boxing even works inside arithmetic expressions

```
Double e = d + 1;
```

- Means:
  - *auto-unbox d into a* `double`
  - *add 1*
  - *auto-box the result into a new* `Double`
  - *store a reference to the newly created wrapper object in e*

ArrayList Question

Suppose data is an `ArrayList<Double>` of size > 0. How do you increment the element with index 0?

Answer: `data.set(0, data.get(0) + 1);`

Comparison

ArrayList<String> myList = new ArrayList<String>();	String[] myList = new String[2];
String a = new String("Whoohoo");	String a = new String("Whoohoo");
myList.add(a);	myList[0] = a;
String b = new String("Frog");	String b = new String("Frog");
myList.add(b);	myList[1] = b;
int theSize = myList.size();	int theSize = myList.length;
String o = myList.get(1);	String o = myList[1];
myList.remove(1);	myList[1] = null;

Lists and beyond...

- Suppose that we want to maintain a list of objects, but without allowing duplicates.
- Can we use a List for this purpose?

  Yes, but...

- It would be nice if there was another, similar class, that does not allow duplicates.
- Java library provides a family of such classes called ***Collection Classes***

Recall our Moveable interface...

```
public class Car implements Moveable {
public void moveBackward() {
System.out.println("Going 95 in reverse");
}

public void moveForward() {
System.out.println("Going 95 on the freeway");

}

}
```


...and Bike and Car classes

```
public class Bike implements Moveable {
public void moveBackward() {
System.out.println("Pedaling backwards!");
}

public void moveForward() {
System.out.println("Pedaling forwards!");

}

}
```

In-Class Exercise II

- 1. Write a method that takes an ArrayList<Moveable> and iterates over it, calling the moveForward() method for each item
- 2. Write a method that takes an ArrayList<Moveable> or an ArrayList of any subclass type of Moveable (e.g. Bike or Car), calling the moveForward() method for each item

Learning Goals Review

- compare and contrast the use of a List over an array
- know how and when to use a List data structure
- compare and contrast the use of generic data structures and arrays of type Object
- compare and contrast assignment with various generic collections under specific subclass scenarios
- use wildcards appropriately in generic type parameters to enable assignment in subclass scenarios