Chapter 7 – Arrays and Array Lists

Declaring and Using Array Lists

- An array list has methods for adding and removing elements in the middle.

- This statement adds a new element at position 1 and moves all elements with index 1 or larger by one position.

```
names.add(1, "Ann")
```

- The remove method,

  removes the element at a given position
  moves all elements after the removed element down by one position
  and reduces the size of the array list by 1.

```
names.remove(1);
```

- To print an array list:

```
System.out.println(names); // Prints [Emily, Bob, Carolyn]
```

Declaring and Using ArrayLists

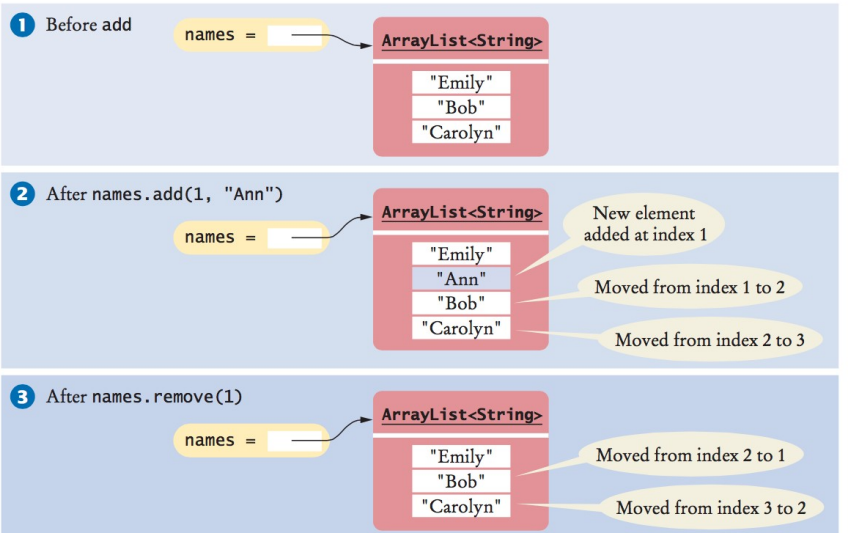


Figure 18 Adding and Removing Elements in the Middle of an Array List

Using the Enhanced for Loop with Array Lists

- You can use the enhanced for loop to visit all the elements of an array list

```
ArrayList<String> names = . . . ;
for (String name : names)
{
   System.out.println(name);
}
```

- This is equivalent to:

```
for (int i = 0; i < names.size(); i++)
{
   String name = names.get(i);
   System.out.println(name);
}
```

Copying Array Lists

- Copying an array list reference yields two references to the same array list.
- After the code below is executed

  Both `names` and `friends` reference the same array list to which the string `"Harry"` was added.

```
ArrayList<String> friends = names;
friends.add("Harry");
```

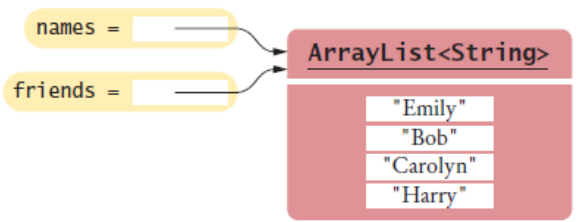


Figure 19 Copying an Array List Reference

- To make a copy of an array list, construct the copy and pass the original list into the constructor:

```
ArrayList<String> newNames = new ArrayList<String>(names);
```

Working with Array Lists

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

Constructs an empty array list that can hold strings.

```
names.add("Ann");
names.add("Cindy");
```

Adds elements to the end.

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

Prints `[Ann, Cindy]`.

```
names.add(1, "Bob");
```

Inserts an element at index 1. `names` is now `[Ann, Bob, Cindy]`.

```
names.remove(0);
```

Removes the element at index 0. `names` is now `[Bob, Cindy]`.

```
names.set(0, "Bill");
```

Replaces an element with a different value. `names` is now `[Bill, Cindy]`.

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

Gets an element at index position i

```
String last =
   names.get(names.size() - 1);
```

Gets the last element.

```
 ArrayList<Integer> squares =
    new ArrayList<Integer>();
 for (int i = 0; i < 10; i++)
{
   squares.add(i * i);
}
```

Constructs an array list holding the first ten squares.

Wrapper Classes

- You cannot directly insert primitive type values into array lists.
- Like truffles that must be in a wrapper to be sold, a number must be placed in a wrapper to be stored in an array list.

- Use the matching wrapper class.

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

Wrapper Classes

- To collect `double` values in an array list, you use an `ArrayList<Double>`.
- if you assign a `double` value to a `Double` variable, the number is automatically "put into a box"
- Called **auto-boxing:**

Automatic conversion between primitive types and the corresponding wrapper classes:

```
Double wrapper = 29.95;
```

Wrapper values are automatically "unboxed" to primitive types

```
double x = wrapper;
```

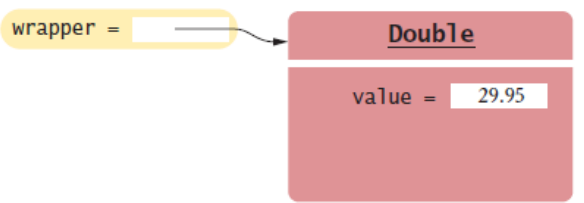


Figure 20 A Wrapper Class Variable

Using Array Algorithms with Array Lists

- The array algorithms can be converted to array lists simply by using the array list methods instead of the array syntax.
- Code to find the largest element in an **array**:

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
   if (values[i] > largest)
   {
      largest = values[i];
   }
}
```

- Code to find the largest element in an **array list**

```
double largest = values.get(0);
for (int i = 1; i < values.size(); i++)
{
   if (values.get(i) > largest)
   {
       largest = values.get(i);
   }
}
```

Choosing Between Array Lists and Arrays

- For most programming tasks, array lists are easier to use than arrays
  - Array lists can grow and shrink.
  - Arrays have a nicer syntax.
- Recommendations
  - If the size of a collection never changes, use an array.
  - If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.
  - Otherwise, use an array list.

Choosing Between Array Lists and Arrays

Table 3 Comparing Array and Array List Operations

Operation	Arrays	Array Lists
Get an element.	`x = values[4];`	`x = values.get(4);`
Replace an element.	`values[4] = 35;`	`values.set(4, 35);`
Number of elements.	`values.length`	`values.size()`
Number of filled elements.	`currentSize` (companion variable, see Section 7.1.4)	`values.size()`
Remove an element.	See Section 7.3.6.	`values.remove(4);`
Add an element, growing the collection.	See Section 7.3.7.	`values.add(35);`
Initializing a collection.	`int[] values = { 1, 4, 9 };`	No initializer list syntax; call `add` three times.

Self Check 7.35

Declare an array list `primes` of integers that contains the first five prime numbers (2, 3, 5, 7, and 11).

Answer:

```
ArrayList<Integer> primes = new ArrayList<Integer>();
primes.add(2);
primes.add(3);
primes.add(5);
primes.add(7);
primes.add(11);
```

Self Check 7.36

Given the array list `primes` declared in Self Check 35, write a loop to print its elements in reverse order, starting with the last element.

Answer:

```
for (int i = primes.size() - 1; i >= 0; i--)
{
   System.out.println(primes.get(i));
}
```

Self Check 7.37

What does the array list names contain after the following statements?

```
ArrayList<String> names = new ArrayList<String>;
names.add("Bob");
names.add(0, "Ann");
names.remove(1);
names.add("Cal");
```

Answer: `"Ann"`, `"Cal"`

Self Check 7.38

What is wrong with this code snippet?

```
ArrayList<String> names;
names.add(Bob);
```

Answer: The `names` variable has not been initialized.

Self Check 7.40

Suppose you want to store the names of the weekdays. Should you use an array list or an array of seven strings?

Answer: Because the number of weekdays doesn't change, there is no disadvantage to using an array, and it is easier to initialize:

```
String[] weekdayNames = { "Monday", "Tuesday",
    "Wednesday", "Thursday", "Friday",
    "Saturday", "Sunday" };
```

Chapter 2 – Using Objects

Copying Object References

- When you copy an object reference

  both the original and the copy are references to the same object

```
Rectangle box = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box;
box2.translate(15, 25);
```

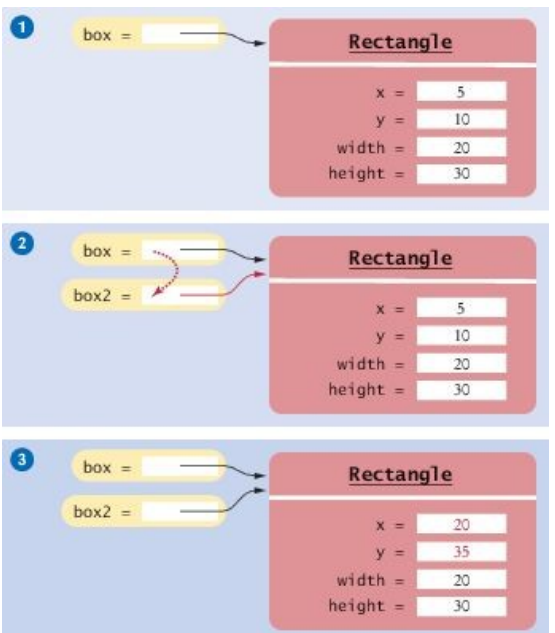


Figure 19 Copying Object References

Self Check 2.38

What is the effect of the assignment `greeting2 = greeting`?

Answer: Now `greeting` and `greeting2` both refer to the same `String` object.

The Public Interface of a Class

The controls of a car form its public interface. The private implementation is under the hood.

- The `String` class declares many other methods besides the `length`. For example, we have `toUpperCase` methods.
- Collectively, the methods **form** the public interface of the class.
- The public interface of a class specifies what you can do with its objects.
- The hidden implementation describes how these actions are carried out.

Chapter 3 - Implementing Classes

Instance Variables and Encapsulation

Figure 1 Tally counter

- Simulator statements:

```
Counter tally = new Counter();
tally.click();
tally.click();
int result = tally.getValue(); // Sets result to 2
```

- Each counter needs to store a variable that keeps track of the number of simulated button clicks.

Instance Variables

- **Instance variables** store the data of an object.
- **Instance of a class:** an object of the class.
- An instance variable is a storage location present in each object of the class.
- The class declaration specifies the instance variables:

```
public class Counter
{
   private int value;
   ...
}
```

- An object's instance variables store the data required for executing its methods.

Instance Variables

- An instance variable declaration consists of the following parts:
  - access specifier (`private`)
  - type of variable (such as `int`)
  - name of variable (such as `myInt`)
- You should declare all instance variables as private.

Instance Variables

- Each object of a class has its own set of instance variables.

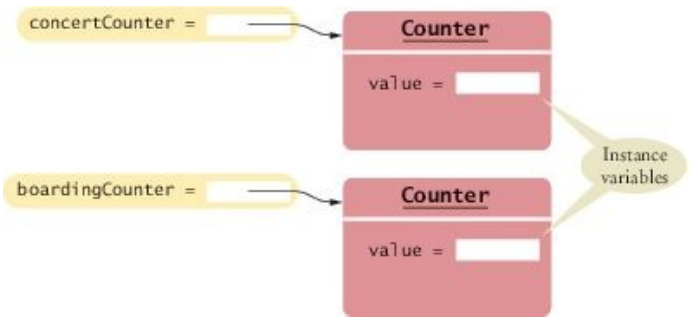
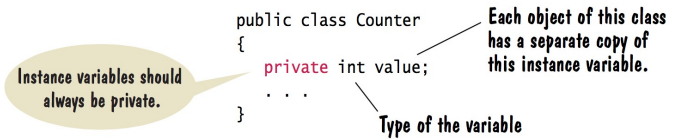


Figure 2 Instance Variables

Syntax 3.1 Instance Variable Declaration

Syntax

```
public class ClassName
{
   private typeName variableName;
   . . .
}
```



Instance Variables

These clocks have common behavior, but each of them has a different state. Similarly, objects of a class can have their instance variables set to different values.

Encapsulation

- **Encapsulation** is the process of hiding implementation details and providing methods for data access.
- To encapsulate data:
  - Declare instance variables as `private` and
  - Declare public methods that access the variables
- Encapsulation allows a programmer to use a class without having to know its implementation.
- Information hiding makes it simpler for the implementor of a class to locate errors and change implementations.

Encapsulation

© yenwen/iStockphoto.

A thermostat functions as a "black box" whose inner workings are hidden.

- When you assemble classes, like `Rectangle` and `String`, into programs you are like a contractor installing a thermostat.
- When you implement your own classes you are like the manufacturer who puts together a thermostat out of parts.

Counter.java

```
1  /**
2  This class models a tally counter.
3  */
4  public class Counter
5  {
6              private int value;
7
8  /**
9  Gets the current value of this counter.
10 @return the current value
11 */
12 public int getValue()
13 {
14             return value;
15 }
16
17 /**
18 Advances the value of this counter by 1.
19 */
20 public void click()
21 {
22             value = value + 1;
23 }
24
25 /**
26 Resets the value of this counter to 0.
27 */
28 public void reset()
29 {
30             value = 0;
31 }
32 }
```

Specifying the Public Interface of a Class

- Methods can also be declared `private`
- `private` methods only be called by other methods in the same class
  `private` methods are not part of the public interface

Specifying Constructors

- Initialize objects
- Set the initial data for objects
- Similar to a method with two differences:
  - The name of the constructor is always the same as the name of the class
  - Constructors have no return type

Specifying Constructors: BankAccount

- Two constructors

```
public BankAccount()
public BankAccount(double initialBalance)
```

- Usage

```
BankAccount harrysChecking = new BankAccount();
BankAccount momsSavings = new BankAccount(5000);
```

BankAccount Public Interface

The constructors and methods of a class go inside the class declaration:

```
public class BankAccount
{
   // private instance variables--filled in  later

   // Constructors
   public BankAccount()
   {
      // body--filled in later
   }
   public BankAccount(double initialBalance)
   {
      // body--filled in later
   }

   // Methods
   public void deposit(double amount)
   {
      // body--filled in later
   }
   public void withdraw(double amount)
   {
      // body--filled in later
   }
   public double getBalance()
   {
      // body--filled in later
   }
}
```

Specifying the Public Interface of a Class

- public constructors and methods of a class form the **public interface** of the class.
- These are the operations that any programmer can use

Syntax 3.2 Class Declaration

Syntax

accessSpecifier `class` *ClassName*
`{`
instance variables
constructors
methods
`}`

```
public class Counter
{
   private int value;

   public Counter(int initialValue) { value = initialValue; }

   public void click() { value = value + 1; }
   public int getValue() { return value; }
}
```

Public interface

Private implementation

Commenting the Public Interface

- Use documentation comments to describe the classes and public methods of your programs.
- Java has a standard form for documentation comments.
- A program called `javadoc` can automatically generate a set of HTML pages.
- Documentation comment

  placed before the class or method declaration that is being documented

Commenting the Public Interface

```
// This is a single line comment


/*
 * This is a regular multi-line comment
 */


/**
 * This is a Javadoc
 */
```

Commenting the Public Interface - Documenting a method

- Start the comment with a `/**`.
- Describe the method's purpose.
- Describe each parameter:

  start with `@param`
  name of the parameter that holds the argument a
  short explanation of the argument

- Describe the return value:

  start with `@return`
  describe the return value

- Omit `@param` tag for methods that have no arguments.
- Omit the `@return` tag for methods whose return type is `void`.
- End with */

Commenting the Public Interface - Documenting a method

- Example:

```
/**
   Withdraws money from the bank account.
   @param amount the amount to withdraw
*/
public void withdraw(double amount)
{
   implementation—filled in later
}
```

- Example:

```
/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   implementation—filled in later
}
```

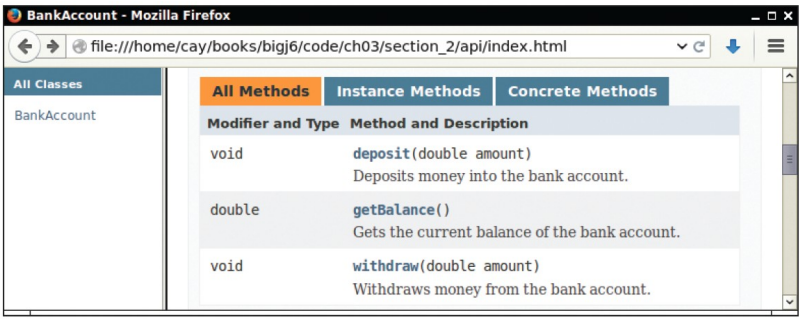
Commenting the Public Interface - Documenting a class

- Place above the class declaration.
- Supply a brief comment explaining the class's purpose.
- Example:

```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   . . .
}
```

- Provide documentation comments for:
  - every class
  - every method
  - every parameter variable
  - every return value

Method Summary: Demo Javadoc



A Method Summary Generated by `javadoc`

Providing Constructors

- Constructor's job is to initialize the instance variables of the object.
- The no-argument constructor sets the balance to zero.

```
public BankAccount()
{
   balance = 0;
}
```

- The second constructor sets the balance to the value supplied as the construction argument.

```
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}
```

Providing Methods

- Is the method an accessor or a mutator

  Mutator method

  - Update the instance variables in some way

  Accessor method

  - Retrieves or computes a result

- `deposit` method - a mutator method

  Updates the balance

```
public void deposit(double amount)
{
   balance = balance + amount;
}
```

Providing Methods -continued

- `withdraw` method - another mutator

```
public void withdraw(double amount)
{
   balance = balance - amount;
}
```

- `getBalance` method - an accessor method

  Returns a value

```
public double getBalance()
{
   return balance;
}
```

Unit Testing

An engineer tests a part in isolation.
This is an example of unit testing.

BankAccountTester.java

```
1  /**
2     A class to test the BankAccount class.
3  */
4  public class BankAccountTester
5  {
6     /**
7        Tests the methods of the BankAccount class.
8        @param args not used
9     */
10    public static void main(String[] args)
11    {
12       BankAccount harrysChecking = new BankAccount();
13       harrysChecking.deposit(2000);
14       harrysChecking.withdraw(500);
15       System.out.println(harrysChecking.getBalance());
16       System.out.println("Expected: 1500");
17    }
18 }
```

Program Run:

```
1500
Expected: 1500
```

Unit Testing - Building a program

- To produce a program: combine both `BankAccount` and `BankAccountTester` classes.
- Details for building the program vary.
- In most environments, you need to carry out these steps:
    1. Make a new subfolder for your program
    2. Make two files, one for each class
    3. Compile both files
    4. Run the test program
- `BankAccount` and `BankAccountTest` have entirely different purposes:

    `BankAccount` class describes objects that compute bank balances

    `BankAccountTester` class runs tests that put a `BankAccount` object through its paces