CN208 Introduction to Computer Programming

Lecture #10

Arrays (Continued) & Exception Handling & Streams

Pimarn Apipattanamontre

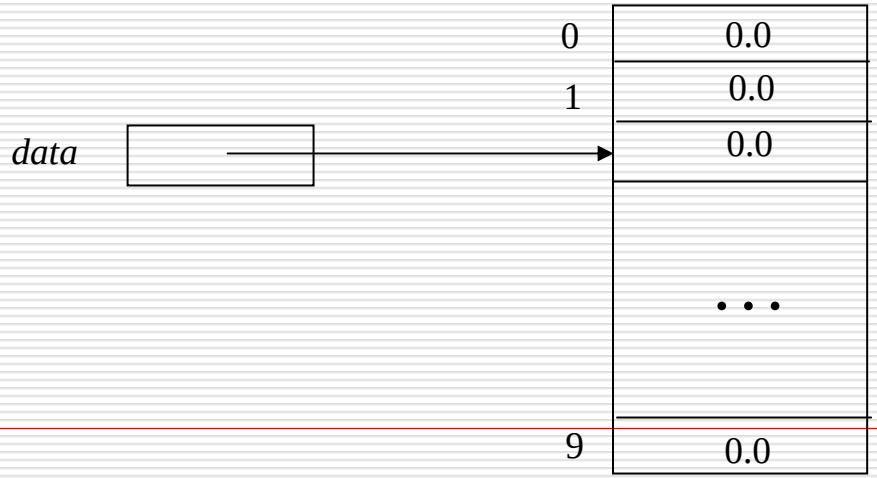
Email: ***pimarn@pimarn.com***

Arrays

- An *array* is a *fixed-length* sequence of values of the same type, which can be an object type or a primitive type.
- To declare an array variable:

```
double[] data = new double[10];
```

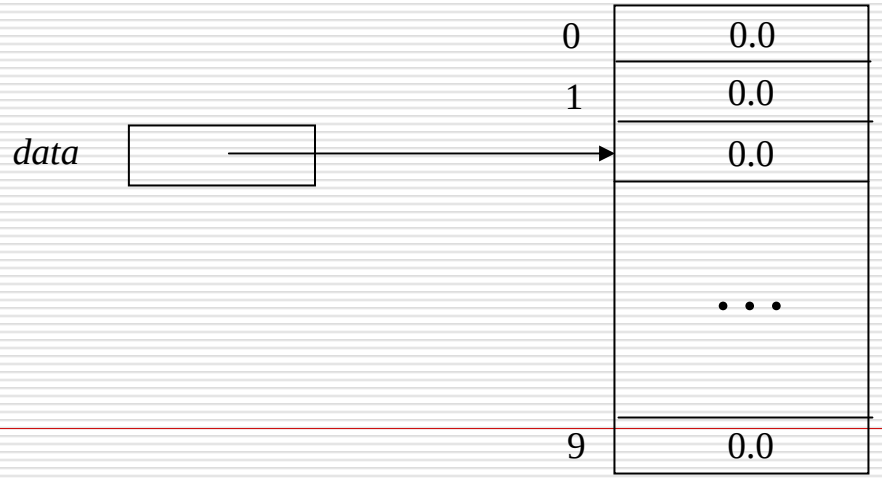
  Creates the actual array of 10 floating-point variables and sets *data* to be a reference to the array.
- When an array is declared, all values are initialized with 0 (for an array of numbers such as int[] or double[]), false (for a boolean[]), or null (for an array of objects).



Arrays

- We access array elements with an integer index:

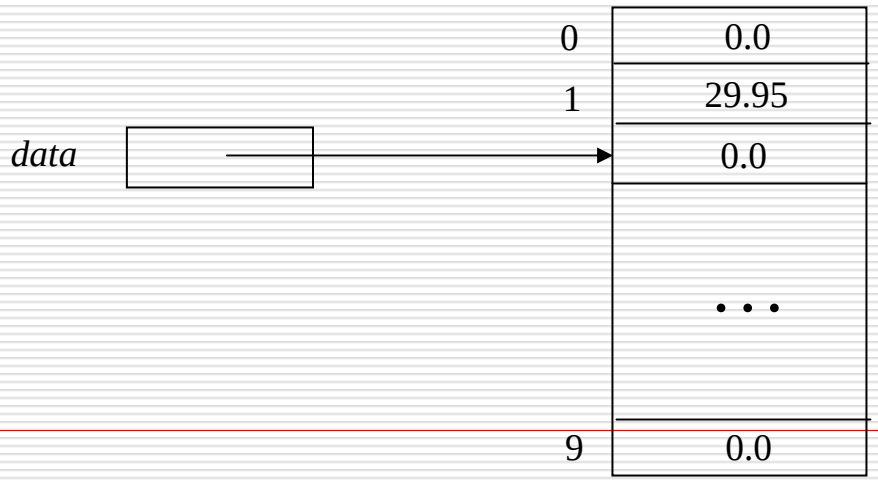
```
data[1] = 29.95;
double x = data[1];
```



Arrays

- We access array elements with an integer index:

```
data[1] = 29.95;
double x = data[1];
```



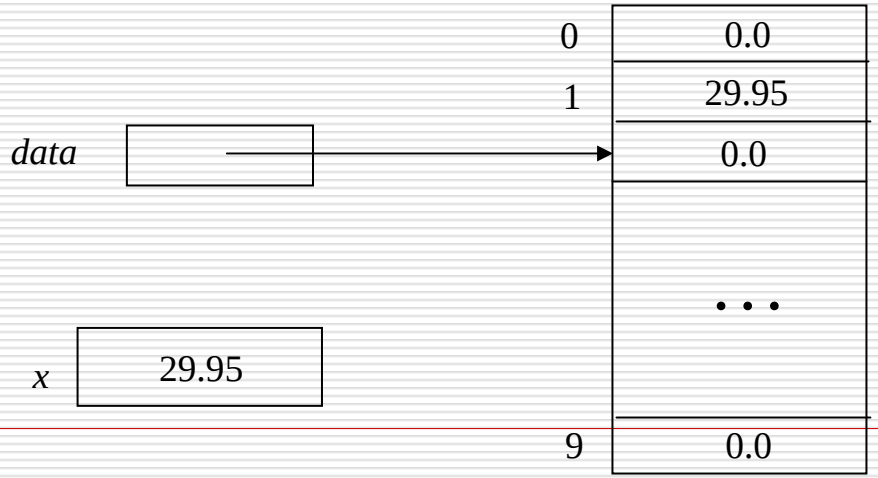
Arrays

- We access array elements with an integer index:

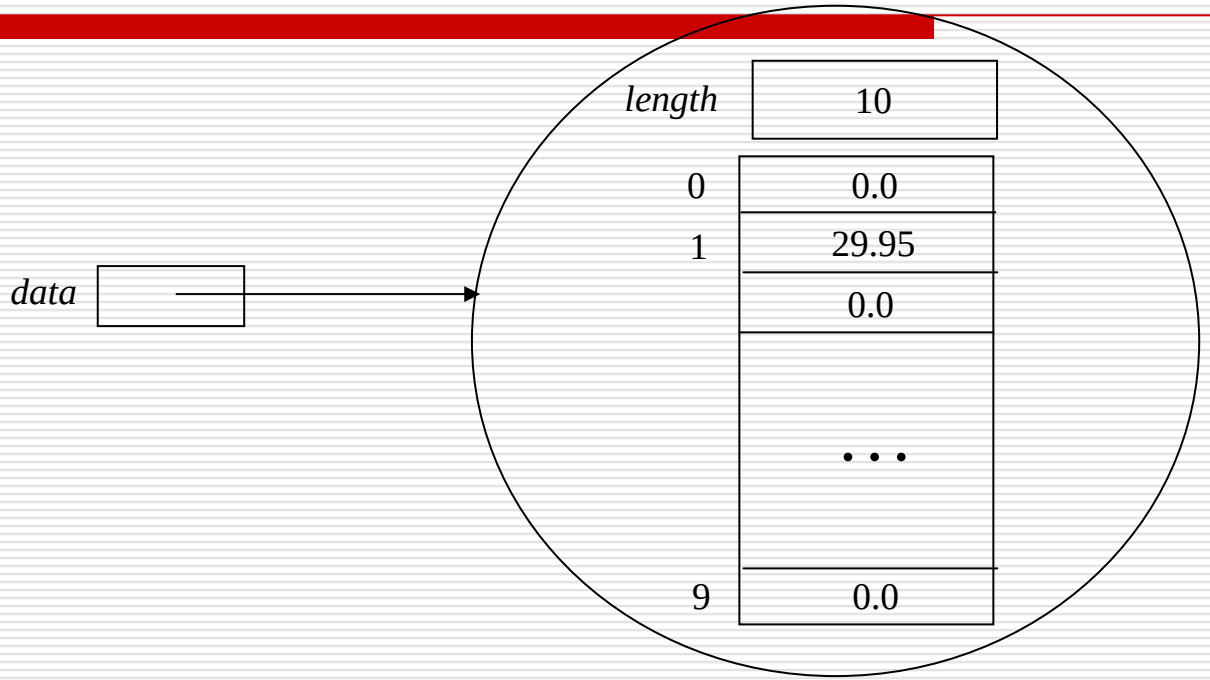
```
data[1] = 29.95;
double x = data[1];
```

- The positions of array elements are numbered starting at 0.
- If we access nonexistent positions, an exception is thrown and the program terminates.

```
int i = 10;
double x = data[i];
```



Arrays



- To find the number of elements in an array, use the *length* field.

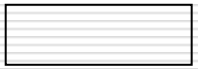
```
double min = data[0];
for (int i = 1; i < data.length; i++)
   if (data[i] < min) min = data[i];
```

- Note that: *length* is a final public instance variable:
  - No parentheses following *length* when using it.
  - We cannot assign a new value to *length*.

Arrays

- To allocate only an array reference (no actual array is created):

```
double[] data;
```

data 

- To create an array and fill each entry:

```
int[] primes = new int[3];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
```

or

```
int[] primes = {2, 3, 5}
```

⇐ Java compiler counts how many elements, allocates an array of correct size, and then fills it with the specified elements.

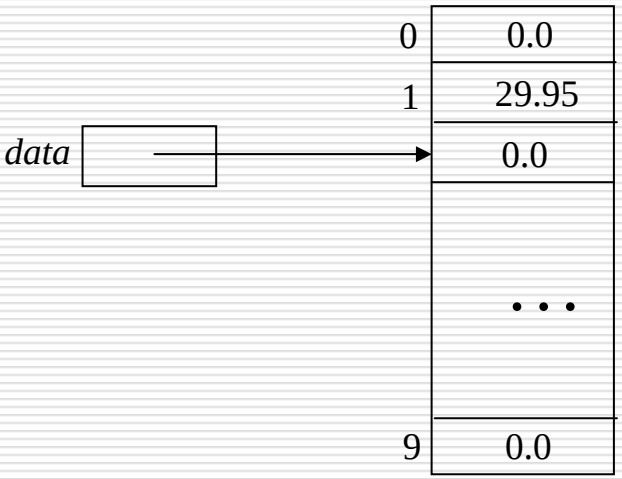
Comparison between Arrays and Array Lists

(1) An array has elements of a specified type while an array list holds a set of references of the *Object* type.

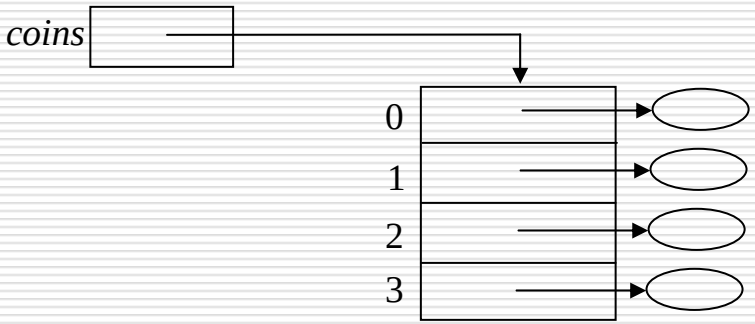
(2) The size of an array is fixed while the array is declared.

The size of an array list varies while an element is inserted or removed.

```
double[] data = new double[10];
data[1] = 29.95;
```




```
ArrayList coins = new ArrayList();
coins.add(...);
coins.add(...);
coins.add(...);
coins.add(...);
```

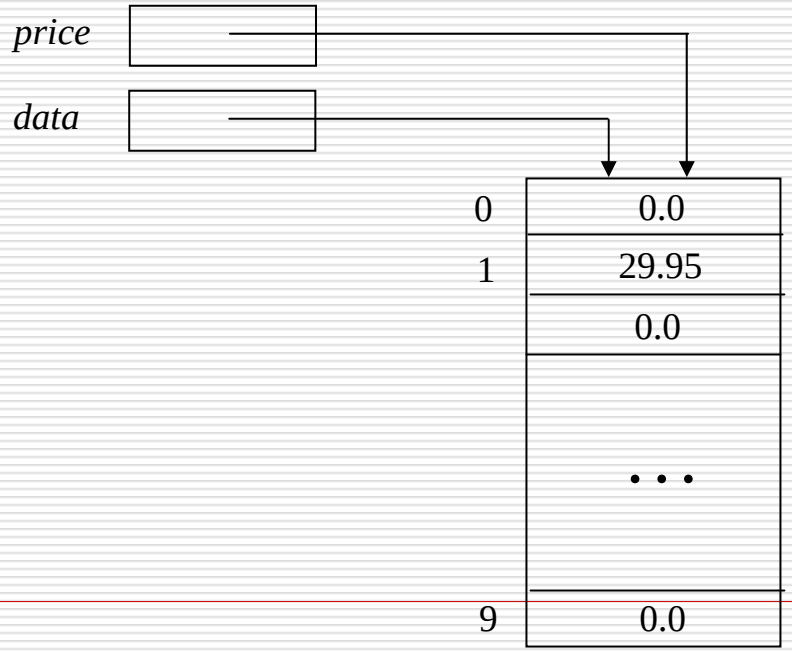


Copying Arrays

- An array variable stores a reference to the array.
- Copying between two array variables causes them referring to the same array.

```
double[] data = new double[10];
data[1] = 29.95;

double[] prices = data;
```

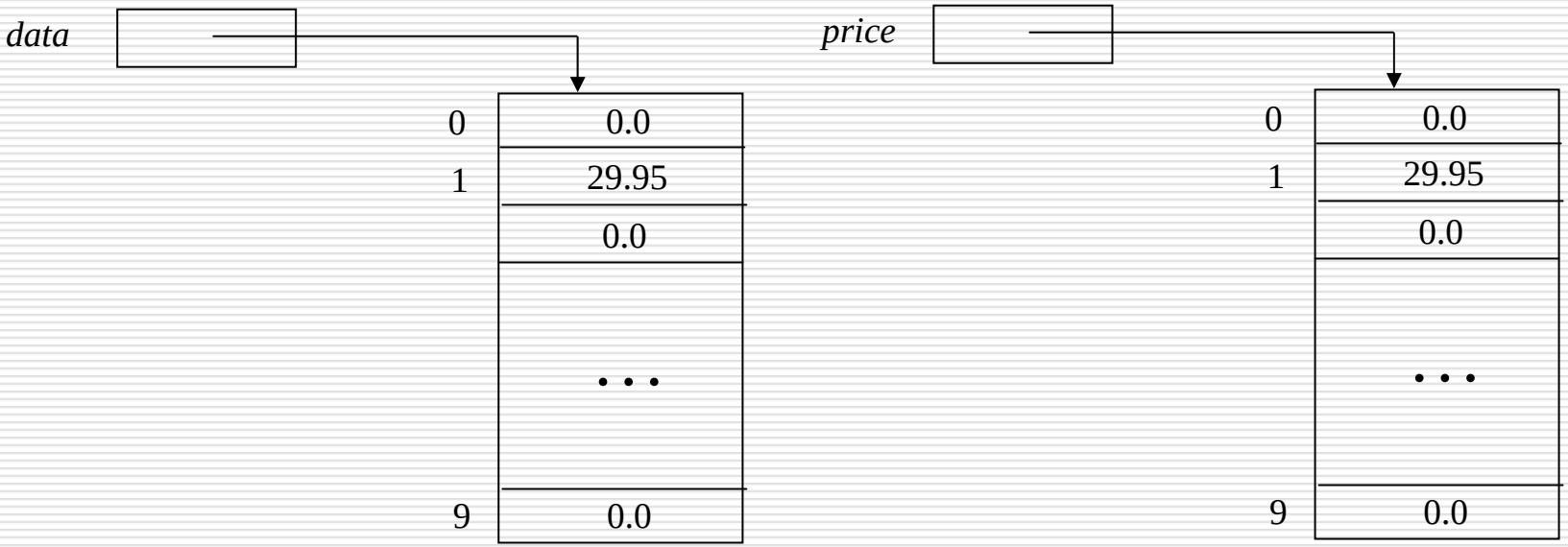


Copying Arrays

- To make a true copy of an array, use the *clone( )* method.

```
double[] prices = (double[]) data.clone();
```

We need to cast the return value of the clone( ) method from the type *Object* to the type *double[]*.

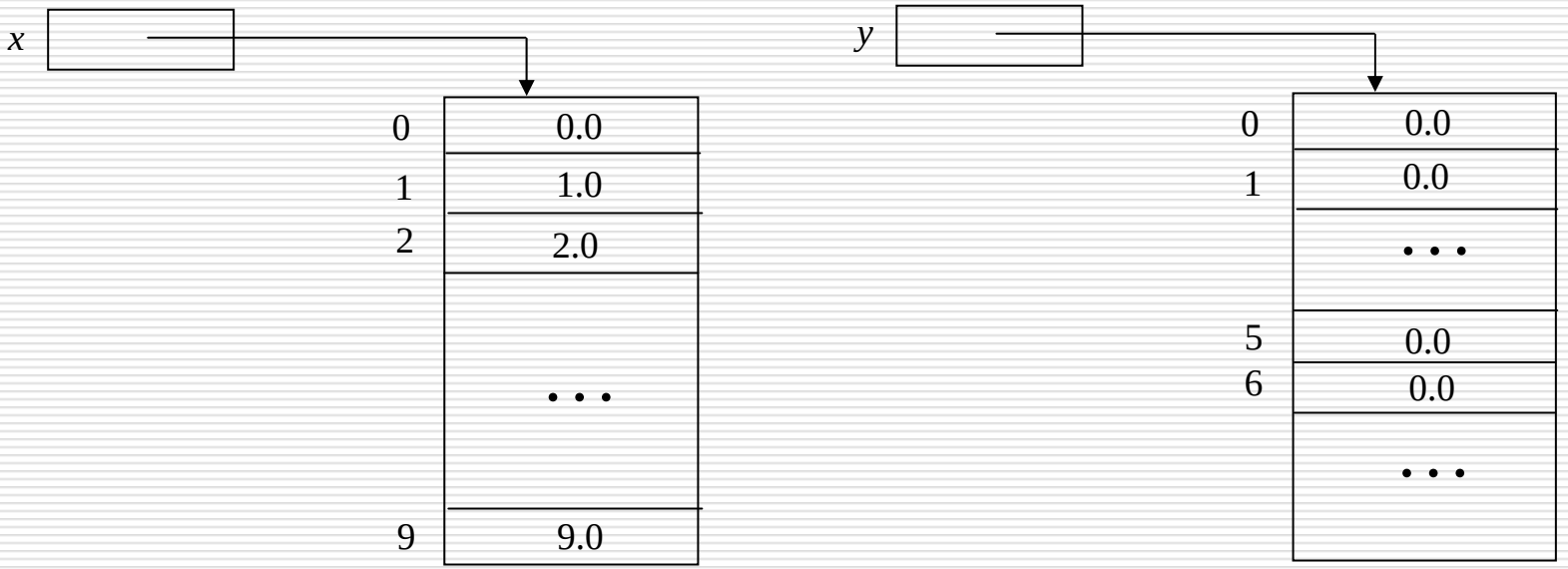


Copying Arrays

- To copy elements from one array to another array (which has the same element type), use:

```
System.arrayCopy(fromArray, fromPos, toArray, toPos, count);
```

Example: *System.arrayCopy(x, 1, y, 5, 2);* ⇦

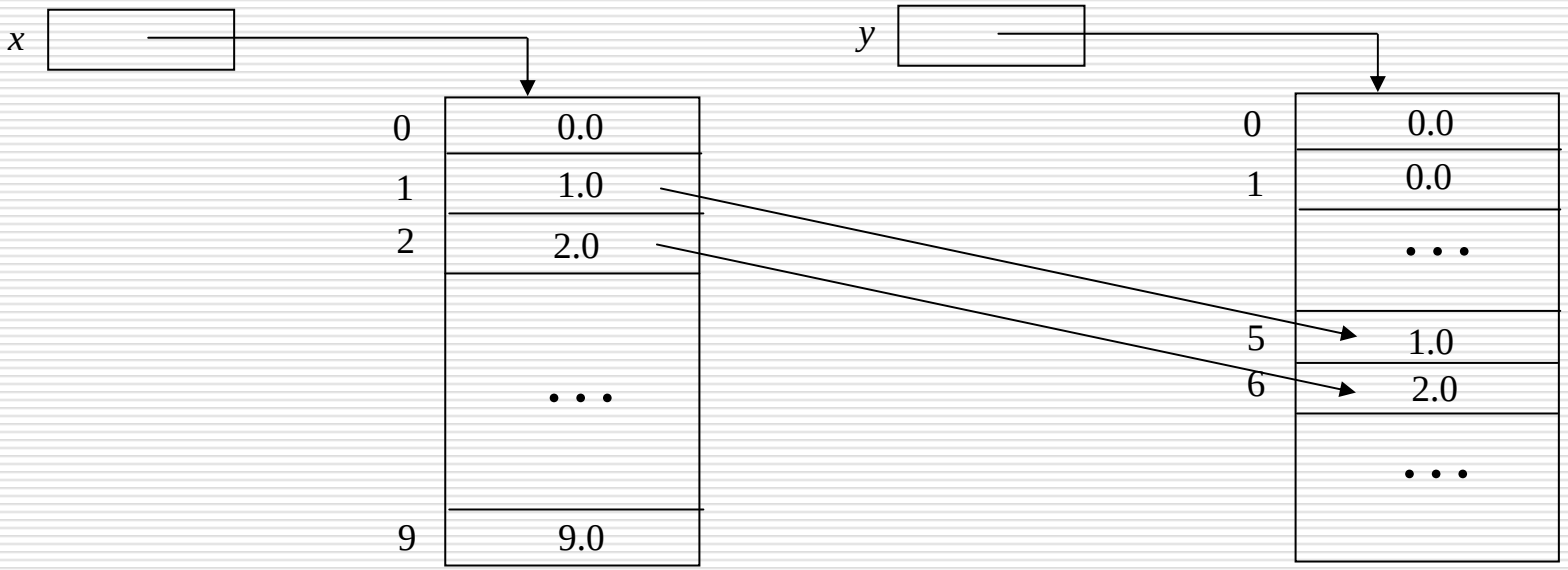


Copying Arrays

- To copy elements from one array to another array (which has the same element type), use:

```
System.arrayCopy(fromArray, fromPos, toArray, toPos, count);
```

Example: *System.arrayCopy(x, 1, y, 5, 2);*



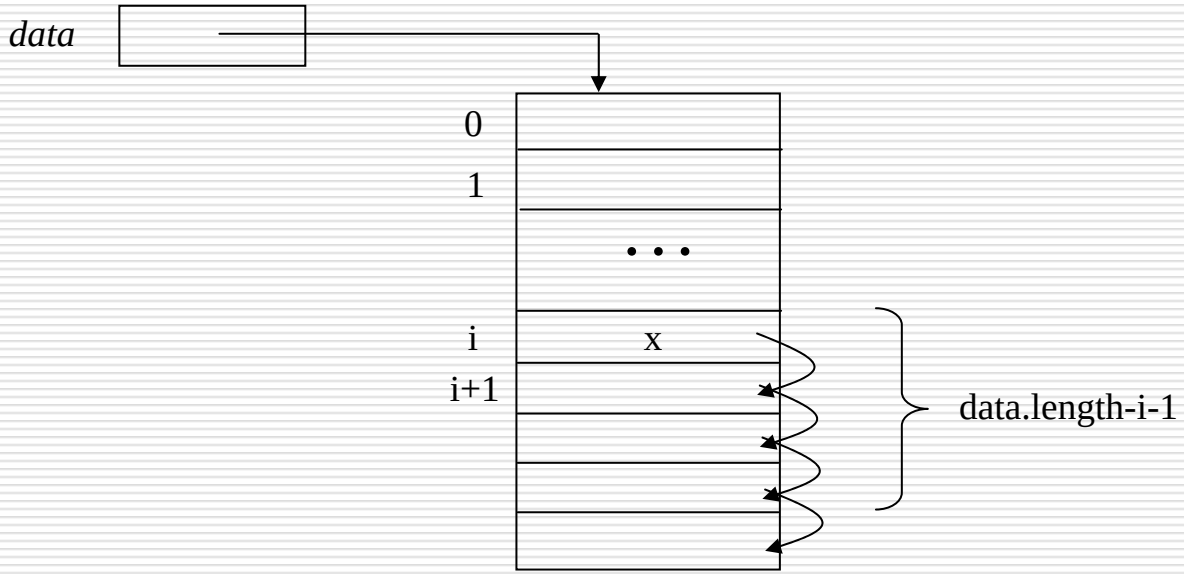
Copying Arrays

Example: `System.arrayCopy(data, i, data, i+1 , data.length-i-1);`

Moves all elements starting from the position *i* one position (the element data[data.length-1] is gone).

Inserts a new element at the position *i*.

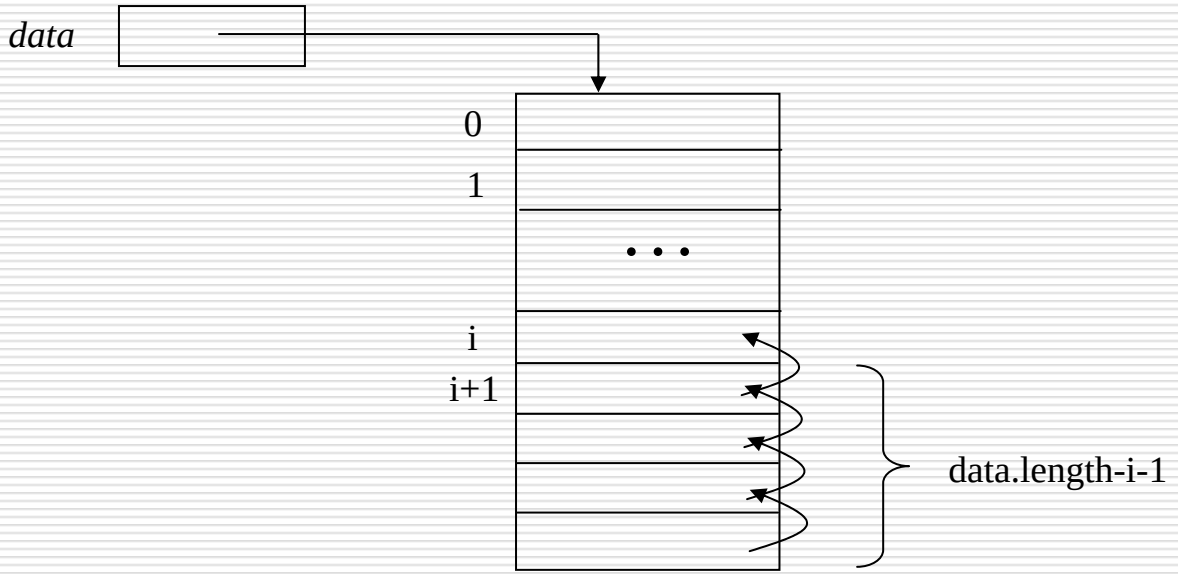
```
data[i] = x;
```



Copying Arrays

Example: `System.arrayCopy(data, i+1, data, i, data.length-i-1);`

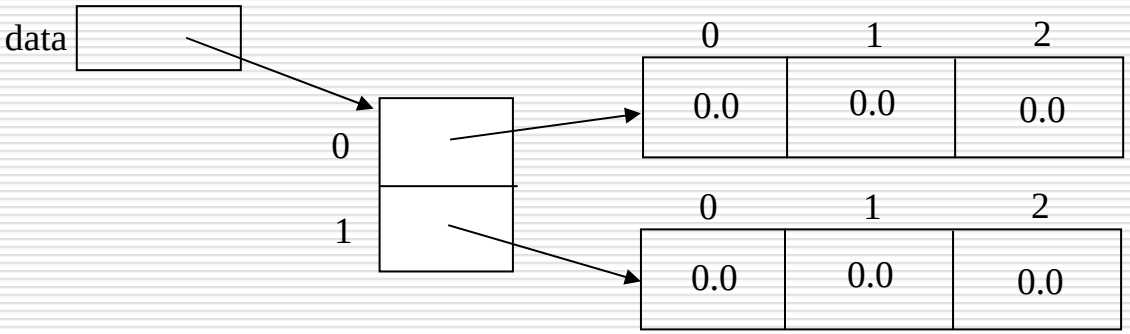
Moves all elements starting from the position i+1 one position (the element data[i] is gone).



Two Dimensional Arrays

- A two-dimensional array forms a table.

```
final int ROWS = 2;
final int COLUMNS = 3;
double[][] data = new double[ROWS][COLUMNS];
```



- To access each element of the two-dimensional array, use two subscript in separate brackets.
- The first one indicates the row number while the second one indicates the column numbers

```
double sum = 0;
for (int i=0; i<ROWS; i++)
  for (int j=0; j<COLUMNS; j++)
    sum = sum + data[i][j];
```

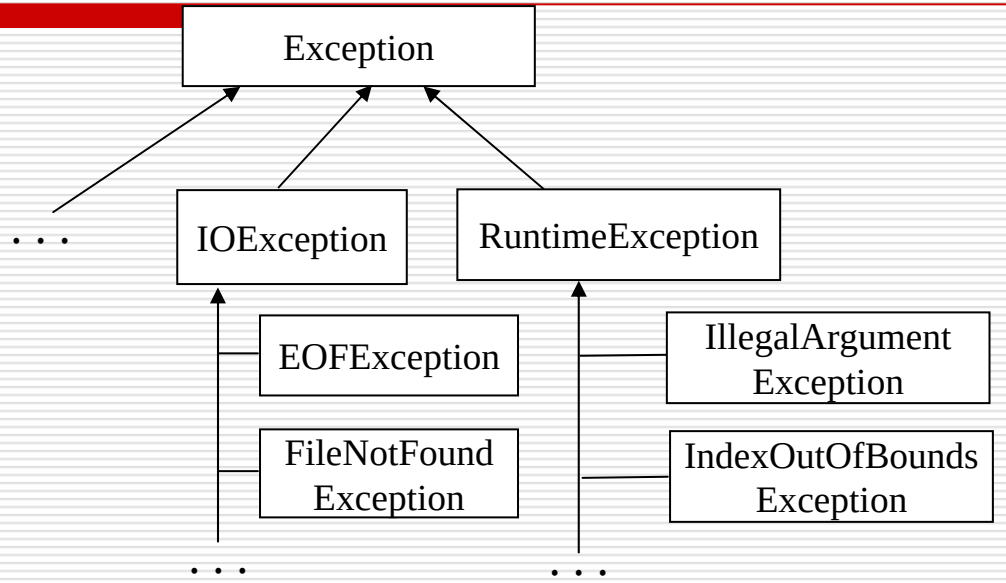
Exception Handling

- An *exception* is a way for a method to indicate an error condition.
- When a method generates an exception, we say that "the method throws an exception".
- We have seen several examples of throwing exceptions in methods of standard Java classes:

  (a) The *get(index)* method of the *ArrayList* class throws *IndexOutOfBoundsException* if index is out of range.

  (b) The *parseInt(s)* method of the *Integer* class throws *NumberFormatException* if the String argument *s* is not an integer string.

Exception Class Hierarchy



- Exception is a class in which exception objects can be created.
- An exception object stores information about the cause of an exception.
- When a method generates an exception, it throws an exception object of the particular exception type.
- The exception object is used for an *exception handler* to handle the error condition.

Throwing an Exception

- We can throw an exception in a method defined in our own class.
- To throw an exception, construct an exception object and use the *throw* statement to throw that exception object.

throw exceptionObject;

```
public class BankAccount
{
  . . .
  public void withdraw(double amount)
  { if (amount > balance)
    { IllegalArgumentException exception
              = new IllegalArgumentException("Amount exceeds balance");
      throw exception;
    }
    balance = balance - amount;
  }
}
```

BankAccount somsakAcc = new BankAccount(100);
somsakAcc.withdraw(200); ⇦ throw IllegalArgumentException and terminate.

- When a method throws an exception, the method returns immediately.
- If there is no *exception handler* to handle the exception, the program terminates.

Catching an Exception

- Instead of terminating the program, we can let the exception handler catch the exception and handle the situation.

```
public class BankAccount
{
  . . .
  public void withdraw(double amount)
  { if (amount > balance)
    { IllegalArgumentException exception
        = new IllegalArgumentException();
      throw exception;
    }
    balance = balance - amount;
  }

  public static final int OVERDRAFT_FEE = 20;
}
```

```
public class TestBankAccount
{
  public static void main(String[] args)
  {   BankAccount anAcc = new BankAccount(100);
      try
      {
        anAcc.withdraw(200);
      }
      catch(IllegalArgumentException e)
      { System.out.println(
        "Withdraw more than available balance");
        anAcc.withdraw(BankAccount.OVERDRAFT_FEE);
      }
      System.out.println(anAcc.getBalance());
      ...
  }
}
```

- To handle exceptions, place statements that can cause the exceptions inside a *try* block and statements handling each type of the exception inside each *catch* block.
- If the exception occurs while executing a method inside the *try* block, an exception object is created and thrown.

Catching an Exception

- Instead of terminating the program, we can let the exception handler catches the exception and handle the situation.

```
public class BankAccount
{
  . . .
  public void withdraw(double amount)
  { if (amount > balance)
    { IllegalArgumentException exception
        = new IllegalArgumentException();
      throw exception;
    }
    balance = balance - amount;
  }

  public static final int OVERDRAFT_FEE = 20;
}
```

```
public class TestBankAccount
{
  public static void main(String[] args)
  {   BankAccount anAcc = new BankAccount(100);
      try
      {
        anAcc.withdraw(200);
      }
      catch(IllegalArgumentException e)
      { System.out.println(
        "Withdraw more than available balance");
        anAcc.withdraw(BankAccount.OVERDRAFT_FEE);
      }
      System.out.println(anAcc.getBalance());
      ...
  }
}
```

- After the exception object is thrown, the *catch* blocks following the *try* block has a chance to catch that exception object.
- The exception object is caught if the type of that object matches the type of parameter of the *catch* blocks.
- Matching the exception object is similar to the case of a method call.

Catching an Exception

- Instead of terminating the program, we can let the exception handler catches the exception and handle the situation.

```
public class BankAccount
{
  . . .
  public void withdraw(double amount)
  { if (amount > balance)
    { IllegalArgumentException exception
        = new IllegalArgumentException();
      throw exception;
    }
    balance = balance - amount;
  }

  public static final int OVERDRAFT_FEE = 20;
}
```

```
public class TestBankAccount
{
  public static void main(String[] args)
  {   BankAccount anAcc = new BankAccount(100);
      try
      {
        anAcc.withdraw(200);
      }
      catch(IllegalArgumentException e)
      { System.out.println(
        "Withdraw more than available balance");
        anAcc.withdraw(BankAccount.OVERDRAFT_FEE);
      }
      System.out.println(anAcc.getBalance()); ⇦ print 80.0
      ...
  }
}
```

- When the exception object is caught in a *catch* block, the statements inside that *catch* block are executed and then execution will proceed with statements following the end of *try-catch* sequence.
- If no exception occurs, the *catch* block is skipped and execution proceeds with statements following the end of *try-catch* sequence.

Catching an Exception

- Instead of terminating the program, we can let the exception handler catches the exception and handle the situation.

```
public class BankAccount
{
  . . .
  public void withdraw(double amount)
  { if (amount > balance)
    { IllegalArgumentException exception
        = new IllegalArgumentException();
      throw exception;
    }
    balance = balance - amount;
  }

  public static final int OVERDRAFT_FEE = 20;
}
```

```
public class TestBankAccount
{
  public static void main(String[] args)
  {   BankAccount anAcc = new BankAccount(100);
      try
      {
        anAcc.withdraw(200);
      }
      catch(IllegalArgumentException e)
      { System.out.println(
        "Withdraw more than available balance");
        anAcc.withdraw(BankAccount.OVERDRAFT_FEE);
      }
      System.out.println(anAcc.getBalance());
      ...
  }
}
```

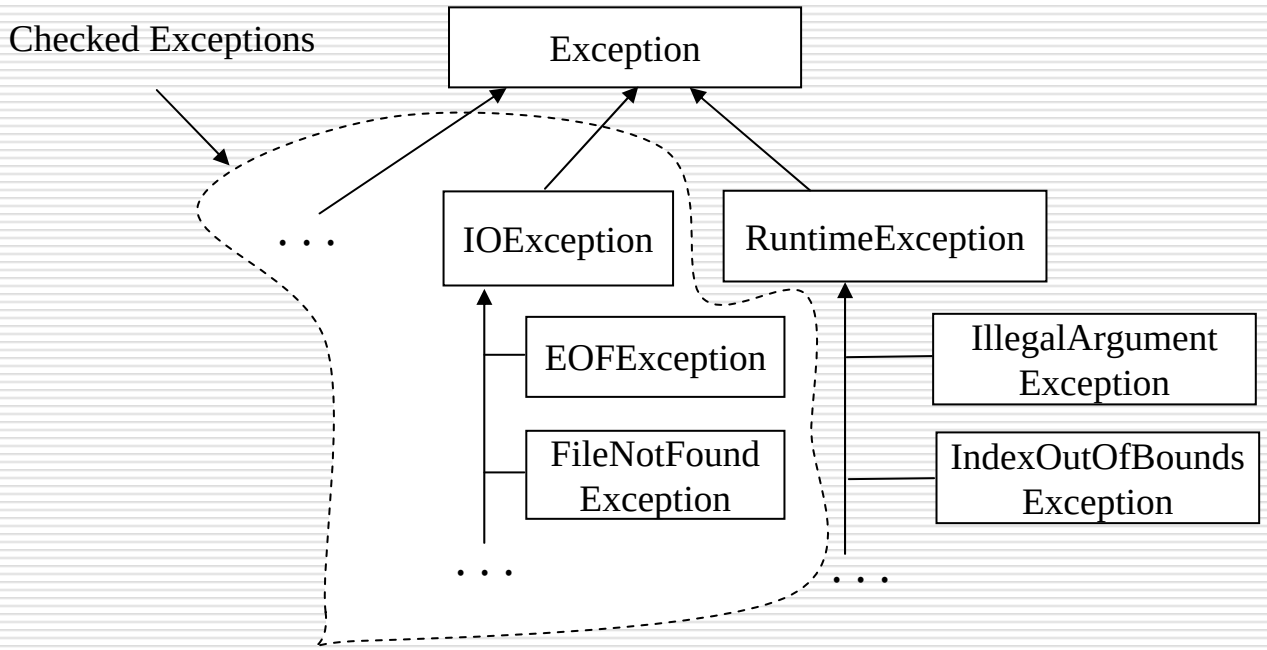
- If no *catch* block can catch the exception object, the current method (in this case the *main* method) returns immediately. Statements following the *try-catch* sequence will not be executed.

Checked Exceptions

- There are two types of exceptions: *Checked* and *Unchecked* exceptions.
- Java compiler checks whether a program handles checked exceptions while it does not check anything for unchecked exceptions.
- Usually the checked exceptions occur when we deal with input and output and they can happen no matter how careful we are.

Checked Exceptions

- All subclasses of *Exception* except the subclass *RuntimeException* are checked exceptions.



Checked Exceptions

- A programmer has a choice whether he/she wants to handle unchecked exceptions but he/she MUST manage checked exceptions explicitly.
- We have two choices to manage a checked exception:

  (a) Throw the checked exception (studied before).

  (b) Let the exception handler handle it (using try-block sequence).

Throw a Checked Exception

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ClassA
{
  . . .
  public void read() throws IOException
  {
    InputStreamReader reader = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(reader);

    String ValueStr = in.readLine();  ⇦ can throw IOException
    . . .
  }
}
```

- We throw a checked exception by tagging the method that contains the method generating a checked exception with a *throws* specifier.

Throw a Checked Exception

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ClassA
{
  . . .
  public void read() throws IOException, ClassNotFoundException
  {
    InputStreamReader reader = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(reader);

    String ValueStr = in.readLine();
    . . .
  }
}
```

⇦ can throw IOException

⇦ Others can throw ClassNotFoundException

- We can tag the method with more than multiple checked exceptions if it contains many methods that can generate many checked exceptions.

Catching Checked and Unchecked Exceptions

```
try
{
  InputStreamReader reader = new InputStreamReader(System.in);
  BufferedReader in = new BufferedReader(reader);
  System.out.println("How old are you?");
  String inputLine = in.readLine();
  int age = Integer.parseInt(inputLine);
  age++;
  System.out.println("Next year, you will be " + age);
}
catch(IOException e)
{
  System.out.println("Input/output error");
}
catch(NumberFormatException e)
{
  System.out.println("Input was not a number");
}
```

- Two exceptions may be thrown: the *readLine( )* method can throw an *IOException* while the *Integer.parseInt( )* method can throw a *NumberFormatException*.
- If either of these exceptions is thrown, the rest of the instructions of the *try* block are skipped, and the appropriate *catch* block is executed.

Checked Exceptions

- If a method calls another method with a *throws* specifier, one of the following can be the case:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ClassA
{
  . . .
  public void read() throws
     IOException, ClassNotFoundException
  {
    InputStreamReader reader =
       new InputStreamReader(System.in);
    BufferedReader in =
       new BufferedReader(reader);

    String ValueStr = in.readLine();
    . . .
  }
}
```

```
import java.io.IOException;

public class ClassB
{
  . . .
  public void methodB()
  {
    ClassA objA = new ClassA();
    try
    {
      objA.read();
      ...
    }
    catch(IOException e) { . . . }
    catch(ClassNotFoundException e) { . . .}
  }
}
```

(1) The method is called from a *try* block which has catch blocks capable of catching *all* possible exception types that may be thrown by the called method. The calling method requires no *throws* specifier.

Checked Exceptions

- If a method calls another method with a *throws* specifier, one of the following can be the case:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ClassA
{
  . . .
  public void read() throws
     IOException, ClassNotFoundException
  {
    InputStreamReader reader =
       new InputStreamReader(System.in);
    BufferedReader in =
       new BufferedReader(reader);

    String ValueStr = in.readLine();
    . . .
  }
}
```

```
import java.io.IOException;

public class ClassB
{
  . . .
  public void methodB() throws IOException
  {
    ClassA objA = new ClassA();
    try
    {
      objA.read();
      ...
    }
    catch(ClassNotFoundException e) { . . .}
  }
}
```

(2) The method is called from a *try* block with catch blocks which catch only <u>*some*</u> of the exceptions that may be thrown. The remaining exception types must appear with *throws* specifier at the calling method.

Checked Exceptions

- If a method calls another method with a *throws* specifier, one of the following can be the case:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ClassA
{
  . . .
  public void read() throws
     IOException, ClassNotFoundException
  {
    InputStreamReader reader =
       new InputStreamReader(System.in);
    BufferedReader in =
       new BufferedReader(reader);

    String ValueStr = in.readLine();
    . . .
  }
}
```

```
import java.io.IOException;

public class ClassB
{
  . . .
  public void methodB()
       throws IOException, ClassNotFoundException
  {
    ClassA objA = new ClassA();
    objA.read();
    ...
  }
}
```

(3) The method is called without a *try* block, all the exception types that may be thrown must appear with throws specifier at the calling method.

The finally Block

- A *finally* block can follow a *try-catch* sequence:

```
try
{
  f();  // call a method that might throw an exception
}
catch(Exception e)
{
  System.out.println(”Calling f failed");
}
finally
{
  g(); // Always call g whatever happens.
}
```

- The *finally* block will always be executed, whether or not an exception is thrown in the *try* block.
- The *finally* block provides a place to put statements that are guaranteed to be executed.

Designing Our Own Exception Types

- We can design our own exception class.

```
public class BankAccount
{
  . . .
  public void withdraw(double amount)
  { if (amount > balance)
    { InsufficientFundsException exception
        = new InsufficientFundsException(
          "Withdrawals of " + amount +
          "exceeds balance of " + balance);
      throw exception;
    }
    balance = balance - amount;
  }
}
```

```
public class InsufficientFundsException
  extends RunTimeException
{
  public InsufficientFundsException()
  {
  }
  public InsufficientFundsException(String reason)
  {
    // construct an exception with the specified
    // detailed message
    super(reason);
  }
}
```

Streams

- There are two types of file formats:
  - Text File Format
    - Data in text files is represented as a sequence of characters.
    - We can read and produce text files easily by using a text editor.
  - Binary Format
    - Data in binary files is represented as a sequence of bytes and it is not in the human-readable form.
    - Binary files is more compact and more efficient.
- All of the classes relating to files are defined in *java.io* package.

Read Data From a Binary File

- To read data from a binary file, we first create a *FileInputStream* object:

  FileInputStream in = new FileInputStream(“input.dat”);

- Use the *read( )* method of the *FileInputStream* class to read a byte of the file.

```
int next = in.read();
byte b;
if (next != -1)
  b = (byte) next;
```

- The *read( )* method returns an *int*, either the byte that was read or the integer -1 if the end of the input stream has been reached.
- We should test the returned value first. If it is not -1 then we can cast it to a byte.

Write Data To a Binary File

- To write data to a binary file, we create a *FileOutputStream* object:

FileOutputStream out = new FileOutputStream(“output.dat”);

- Use the *write( )* method of the *FileOutputStream* class to write the specified byte *b*.

out.write(b);

- Use the close*( )* method to close all files we no longer need.

in.close( );

out.close( );

Read Data From a Text File

- To read data from a text file, we first create a *FileReader* object:

FileReader reader = new FileReader("input.txt");

- Use the *read( )* method of the *FileReader* class to read a single character from a text file.

```
int next = reader.read();
char c;
if (next != -1)
  c = (char) next;
```

- The *read( )* method returns an *int*, either the character that was read or the integer -1 if the end of the input stream has been reached.
- We should test the returned value first. If it is not -1 then we can cast it to a char.

Write Data To a Text File

- To write data to a text file, create a *FileWriter* object:

FileWriter writer = new FileWriter(“output.txt”);

- Use the *write( )* method of the *FileWriter* class to write the specified character *c*.

writer.write(c);

- Similar to the binary files, we use the close*( )* method to close all text files.

reader.close( );

writer.close( );

Reading and Writing Text Files

- The *write( )* method of the *FileWriter* class sends output to a file one character at a time.

FileWriter writer = new FileWriter("output.txt");

writer.write('A');

- To print numbers, objects, or strings, we can construct a *PrintWriter* from a *FileWriter* object.

PrintWriter out = new PrintWriter(writer);

- Then use the *print( )* or *println( )* method to send the output to the file.

out.println(29.95);

out.println("Hello, World!");

BankAccount anAcc = new BankAccount(100);

out.println(anAcc);

Reading and Writing Text Files

```
FileWriter writer = new FileWriter("output.txt");
PrintWriter out = new PrintWriter(writer);
out.println(29.95);
out.println("Hello, World!");
BankAccount anAcc = new BankAccount(100);
out.println(anAcc);
```

- The *print( )* and *println( )* method:

  (a) Convert a number to the string representation or use the *toString( )* method to convert an object to the string.

  (b) Break up the string into individual characters and then give each character to the *FileWriter* object through its *write( )* method.