Java:

Learning to Program with Robots

Chapter 09: Input and Output

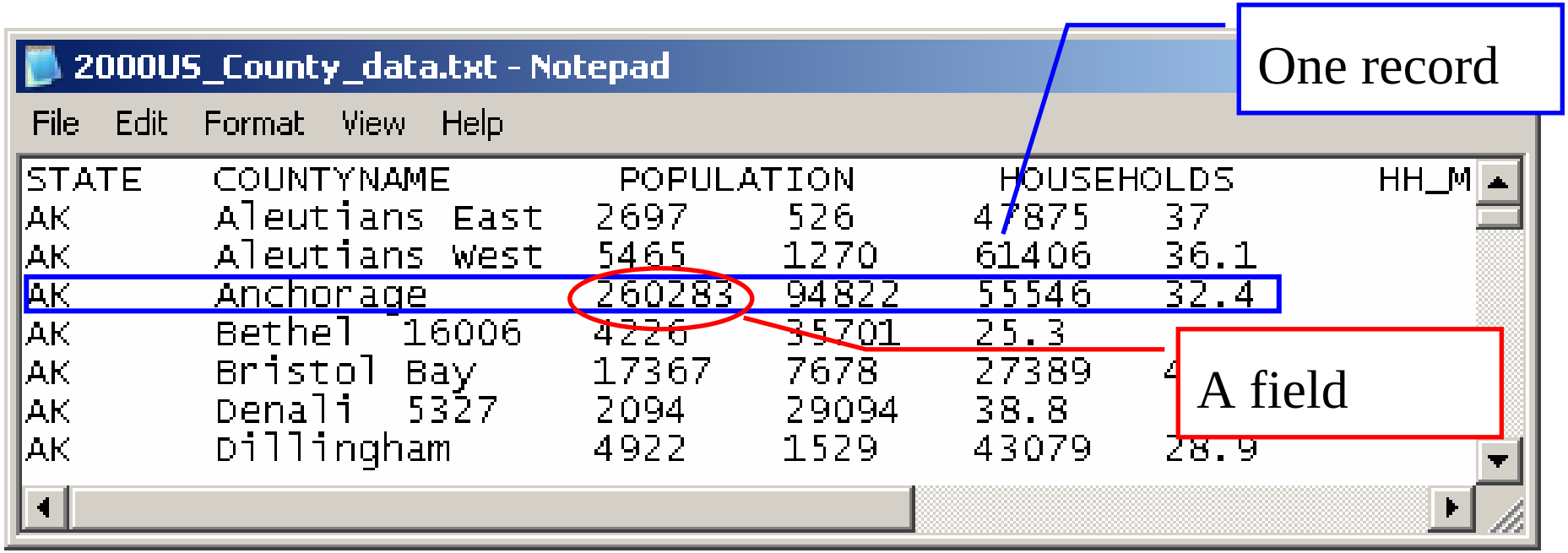
Chapter Objectives

After studying this chapter, you should be able to:

- Use the **Scanner** class to read information from files and the **PrintWriter** class to write information to files
- Locate files using absolute and relative path
- Use the **Scanner** class to obtain information from the user
- Prompt users for information and check it for errors
- Give users more control over programs by using command interpreters
- Put commonly used classes in a package
- Use a dialog box to obtain a filename from the user
- Display an image stored in a file

Files store information.

- Might be unstructured (like a letter)
- Might be structured into records, with each record consisting of many fields.



9.1.1: Reading from a File

Most file-processing programs have three important steps:

- Locate the file on the disk and construct objects used to access it.
- Process the file, one record at a time.
- Close the file after the program has finished using it.

```
public class ReadCountyData
{
   public static void main(String[ ] args)
   {  Scanner in = new Scanner(location of input file);

      while (the file has another record)
      {  read the record
         process the information in the record
      }

      in.close();
   }
}
```

9.1.1: Example of Reading from a File (1/2)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/** Read census data, printing all those lines containing the string "AK" (Alaska).
*
* @author Byron Weber Becker */
public class ReadCountyData
{
   public static void main(String[ ] args)
   {  // Open the file.
      Scanner in = null;
      try
      {  File file = new File("2000US_County_data.txt");
         in = new Scanner(file);
      } catch (FileNotFoundException ex)
      {  System.out.println(ex.getMessage());
         System.out.println("in " + System.getProperty("user.dir"));
         System.exit(1);
      }
```

9.1.1: Example of Reading from a File (2/2)

```
        // Read and process each record.
        while (in.hasNextLine())
        {  String record = in.nextLine();
           if (record.indexOf("AK") >= 0)      // records containing “AK” (Alaska)
           {  System.out.println(record);
           }
        }

        // Close the file.
        in.close();
    }
}
```

9.1.2: Example of Writing to a File (1/2)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/** Read census data, writing all those lines containing the string "AK" (Alaska) to a file.
*
* @author Byron Weber Becker */
public class WriteMatchingLines
{
   public static void main(String[ ] args)
   {  // Open the files.
      Scanner in = null;
      PrintWriter out = null;
      try
      {  in = new Scanner(new File("2000US_County_data.txt"));
         out = new PrintWriter("Alaska.txt");
      } catch (FileNotFoundException ex)
      {  System.out.println(ex.getMessage());
         System.out.println("in " + System.getProperty("user.dir"));
         System.exit(1);
      }
```

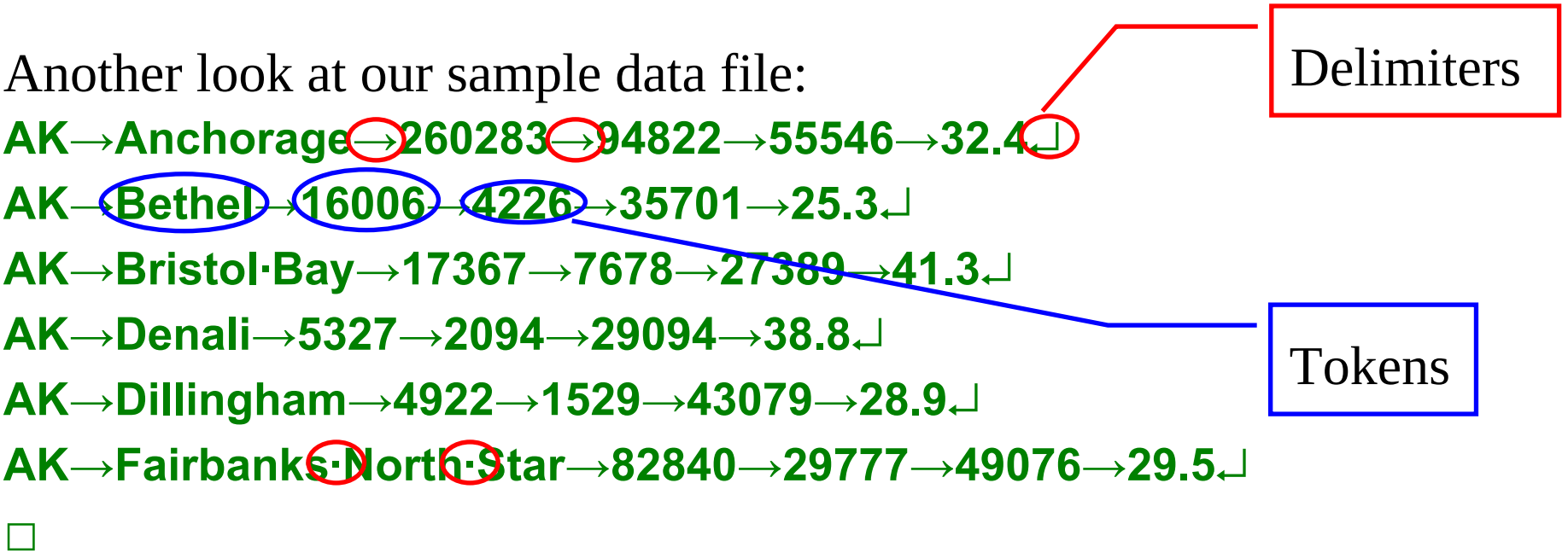
9.1.2: Example of Writing to a File (2/2)

```
        // Read and process each record.
        while (in.hasNextLine())
        {  String record = in.nextLine();
           if (record.indexOf("AK") >= 0)
           {  out.println(record);
           }

        }

        // Close the files.
        in.close();
        out.close();
    }
}
```

A text file contains a sequence of characters.

- Some are obvious: **a**, **b**, **c**, and **1**, **2**, **3**.
- Some are less obvious: spaces, tabs, end of line (we'll show these with ·, →, and ↵, respectively)
- A useful fiction is to think of the end of the file as marked with another character that we'll show as □.

Another look at our sample data file:



The **Scanner** class contains a number of methods to read a token from the file and convert it into an appropriate type. For example:

File:

AK→Anchorage→260283→94822→55546→32.4↵

Corresponding Code:

```
while (in.hasNextLine())
{  String record = in.nextLine();
   String stAb = in.next();            // state abbreviation
   String cNm = in.next();             // county name
   int pop = in.nextInt();             // population
   int hh = in.nextInt();              // number of households
   int inc = in.nextInt();             // median income
   double age = in.nextDouble();       // median age
   in.nextLine();

   if (stAb.equals("AK"))
   {  System.out.println(stAb + " " + cNm + " " + pop + " "
                         + hh + " " + inc + " " + age);
   }
}
```

9.1.3: Tracing Data Acquisition Methods

Statement Input	stAb	cNm	inc	age
♦AK→Anchorage→260283→94822→55546→32.4↵				
while (in.hasNextLine())				
♦AK→Anchorage→260283→94822→55546→32.4↵				
String stAb = in.next();	AK			
AK♦→Anchorage→260283→94822→55546→32.4↵				
String cNm = in.next();	AK	Anchorage		
AK→Anchorage♦→260283→94822→55546→32.4↵				
…				
AK→Anchorage→260283→94822→♦55546→32.4↵				
int inc = in.nextInt();	AK	Anchorage	55546	
AK→Anchorage→260283→94822→55546→♦32.4↵				
double age = in.nextDouble();	AK	Anchorage	55546	32.4
AK→Anchorage→260283→94822→55546→32.4♦↵				
in.nextLine();	AK	Anchorage	55546	32.4
AK→Anchorage→260283→94822→55546→32.4↵				
♦AK→Bethel→16006→4226→35701→25.3↵				
while (in.hasNextLine())	AK	Anchorage	55546	32.4
♦AK→Bethel→16006→4226→35701→25.3↵				
String stAb = in.next();	AK	Anchorage	55546	32.4

Suppose the average age was sometimes unavailable, replaced by **NA**.

AK→Anchorage→260283→94822→55546→32.4↵

AK→Bethel→16006→4226→35701→NA↵

AK→Bristol·Bay→17367→7678→27389→41.3↵

Our previous program would throw **InputMismatchException** when it tried to read **NA** with **in.nextDouble()**.

Instead, use:

```
int inc = in.nextInt();        // median income

double age;
if (in.hasNextDouble())
{  age = in.nextDouble();
} else
{  age = -1;          // Flag that data is not available
   in.next();         // Skip over "NA"
}
```

Other methods:

- **hasNext()**
- **hasNextInt()**
- **hasNextDouble()**
- **hasNextBoolean()**
- **hasNextLine()**

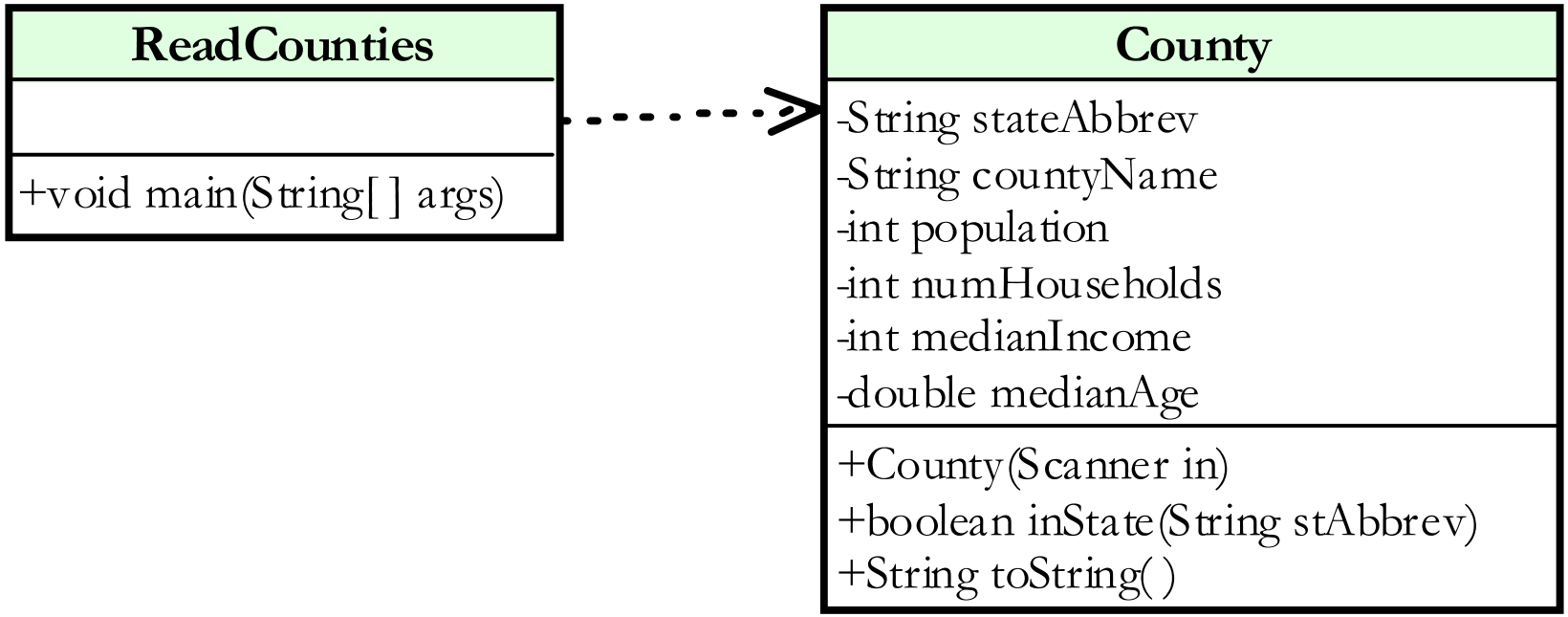
All return **boolean** values.

We can also use data availability methods to solve the problem of multi-word county names such as “Fairbanks North Star”:

```
// Read and process each record.
in.nextLine();                          // skip header record
while (in.hasNextLine())
{  String stAb = in.next();             // state abbreviation

   String cNm = in.next();              // County name – might consist of several tokens
   while (!in.hasNextInt())
   {  cNm += " " + in.next();
   }

   int pop = in.nextInt();              // population
   int hh = in.nextInt();               // number of households
   int inc = in.nextInt();              // median income
   double age = -1;                     // median age – might be NA
   if (in.hasNextDouble())
   {  age = in.nextDouble();
   }  else
   {  in.next();
   }
   in.nextLine();
}
```

- Reading a record from a file can get complex!
- A single record is an excellent candidate for an object.

Therefore, write a class to represent the information in the records. Initialize the object by writing a constructor that reads from the file.



Old Way…

```
// Read and process each record.
in.nextLine();       // skip header record
while (in.hasNextLine())
{  String stAb = in.next();

   String cNm = in.next();
   while (!in.hasNextInt())
   {   cNm += " " + in.next();
   }

   int pop = in.nextInt();
   int hh = in.nextInt();
   int inc = in.nextInt();
   double age = -1;
   if (in.hasNextDouble())
   {   age = in.nextDouble();
   }   else
   {   in.next();
   }
   in.nextLine();

   if (stNm.equals("AK"))
   {   System.out.println(stAb + " " + cNm
       + " " + pop + " "  + hh + " " + inc
       + " " + age);
   }
}
```

New (and Better) Way…

```
// Read and process each record.
in.nextLine();          // skip header record
while (in.hasNextLine())
{  County c = new County(in);

   if (c.inState("AK"))
   {  System.out.println(c);
   }
}
```

9.2.1: Reading Records as Objects (2/3)

```
import java.util.Scanner;

public class County extends Object
{  private String stateAbbrev;
   private String countyName;
   private int population;
   private int numHouseholds;
   private int medianIncome;
   private double medianAge;

   /** Read one record from a file to initialize a new instance of County.
   * @param in The open input file with the file cursor positioned just before the record to read. */
   public County(Scanner in)
   {  super();
      this.stateAbbrev = in.next(); // state abbreviation
      this.countyName = in.next();
      while (!in.hasNextInt())
      {  this.countyName += " " + in.next(); // county name
      }
      this.population = in.nextInt(); // population
      this.numHouseholds = in.nextInt(); // number of households
      this.medianIncome = in.nextInt(); // median income
      this.medianAge = in.nextDouble(); // median age
      in.nextLine();
   }
```

```
    /** Is this county in the specified state? */
    public boolean inState(String stAbbrev)
    {  return this.stateAbbrev.equals(stAbbrev);
    }

    /** Represent this county as a string. */
    public String toString()
    {  return this.stateAbbrev + " " + this.countyName + " " + this.population
            + " " + this.numHouseholds + " " + this.medianIncome + " "
            + this.medianAge;
    }
}
```

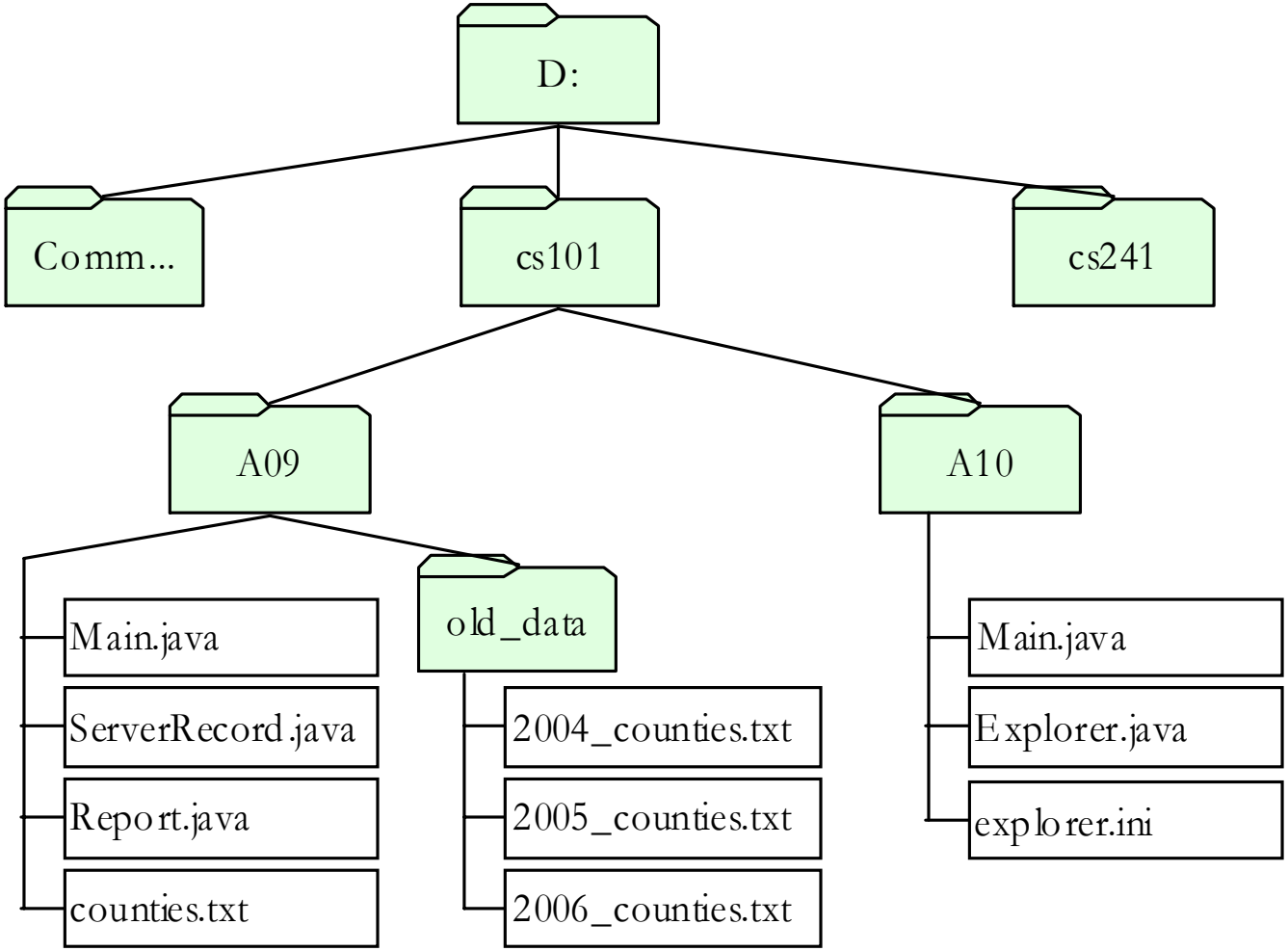
Consider two different file formats and the code to read it:

File Format	*Code*

AK→Anchorage→260283↵
AK→Fairbanks North Star→82840↵
AK→Juneau→30711↵
	this.stateAbbrev = in.next();
	this.countyName = in.next();
	while (!in.hasNextInt())
	{ this.countyName +=
	" " + in.next();
	}
	this.population = in.nextInt();

AK→Anchorage↵
260283↵
AK→Fairbanks North Star↵
82840↵
AK→Juneau↵
30711↵
	this.stateAbbrev = in.next();
	this.countyName =
	in.nextLine().trim();
	this.population = in.nextInt();

Valid file names and locations:




```
File f = new File("D:/cs101/A09/counties.txt");
File f = new File("D:/cs101/A09old_data/2006_counties.txt");
File f = new File("counties.txt");
File f = new File("old_data/2005_counties.txt");
File f = new File("../A10/explorer.ini");
System.out.println(System.getProperty("user.dir"));
```

9.3.3: Manipulating Files

A **File** object represents a path to a file or directory that may or may not exist. Methods provided include:

boolean canRead()	Can this program read the file?
boolean canWrite()	Can this program write to the file?
boolean delete()	Delete the file or directory. Directories must be empty. Returns **true** if it succeeds.
boolean exists()	Returns **true** if the file exists.
String getAbsolutePath()	Gets the absolute path for this file.
File getParentFile()	Get a **File** object associated with this file's parent directory.
boolean isFile()	Does the path specifies a file?
boolean isDirectory()	Does the path specifies a directory?
long length()	Gets the number of characters in the file.
void mkDir()	Makes the directory represented by this **File**. Returns **true** if successful.

Suppose we had many data files and knew the user was curious about the median age of the residents. We could implement code according to the following pseudocode:

```
ask the user for the data file to open
String fileName = get data file's name
ask the user for the maximum age
double maxAge = get the maximum age

open the data file named in fileName
while (data file has another line)
{  County c = new County(data file);
   if (c.getMedianAge() <= maxAge)
   {  print the record
   }
}
close the data file
```

9.4.1: Reading from the Console

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/** Read census data from a user-specified file.
*
* @author Byron Weber Becker */
public class UserInput
{
   public static void main(String[ ] args)
   {  // Open the console
      Scanner cin = new Scanner(System.in);

      System.out.print("County file name: ");
      String fName = cin.nextLine().trim();

      System.out.print("Maximum age:  ");
      double maxAge = cin.nextDouble();

      // Open the file.
      Scanner in = null;
      try
      {  File file = new File(fName);
         in = new Scanner(file);
```

9.4.2: Checking Input for Errors (1/3)

We could replace:

```
System.out.print("Maximum age:  ");
double maxAge = cin.nextDouble();
```

with:

```
int maxAge = 0;
while (true)
{  System.out.print("Maximum age:  ");
   if (cin.hasNextDouble())
   {  maxAge = cin.nextDouble();
      cin.nextLine();          // Consume remaining input
      break;
   } else
   {  String next = cin.nextLine();          // Consume erroneous input
      System.out.println(next + " is not age such as 24 or 39.3.");
   }
}
```

9.4.2: Checking Input for Errors (2/3)

```
public class Prompt extends Object
{
   private static final Scanner cin = new Scanner(System.in);

   public static double forDouble(String prompt)
   {  while (true)
      {  System.out.print(prompt);
         if (Prompt.cin.hasNextDouble())
         {  double answer = Prompt.cin.nextDouble();
            Prompt.cin.nextLine();          // Consume remaining input
            return answer;
         }  else
         {  String input = Prompt.cin.nextLine();
            System.out.println("Error:  " + input + " not recognized.");
         }
      }
   }
}
```

To use, replace the following code (which does not check its input):

```
System.out.print("Maximum age:  ");
double maxAge = cin.nextDouble();
```

with:

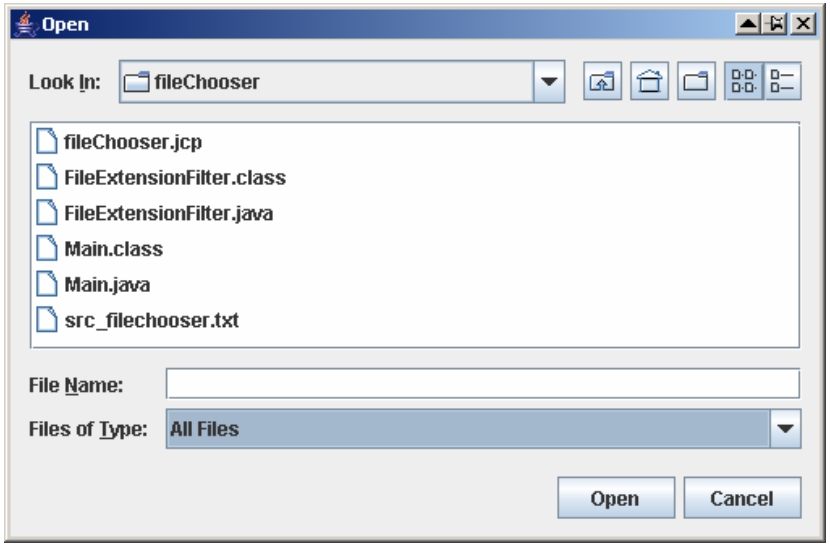
```
double maxAge = Prompt.forDouble("Maximum age: ");
```

9.4.2: Checking Input for Errors (3/3)

```
public class Prompt extends Object
{  private static final Scanner cin = new Scanner(System.in);

   public static double forDouble(String prompt) …

   public static File forInputFile(String prompt)
   {  while (true)
      {  System.out.print(prompt);
         String name = Prompt.cin.nextLine().trim();
         File f = new File(name);
         if (!f.exists())
         {  System.out.print("Error:  " + name + " does not exist.");
         } else if (f.isDirectory())
         {  System.out.print("Error:  " + name + " is a directory.");
         } else if (!f.canRead())
         {  System.out.print("Error:  " + name + " is not readable.");
         } else
         {  return f;
         }
      }
   }

   public static Scanner forInputScanner(String prompt)…
```

9.8.1: Using JFileChooser

```
import javax.swing.JFileChooser;
import java.io.File;

public class Main extends Object
{
   public static void main(String[ ] args)
   {  System.out.println(
                  "Ready to get a file name.");

      JFileChooser chooser = new JFileChooser();
      chooser.setCurrentDirectory(new File(System.getProperty("user.dir")));

      // show the dialog – program stops until the user presses one of the buttons
      int result = chooser.showOpenDialog(null);
      if (result == JFileChooser.APPROVE_OPTION)
      {  System.out.println("You chose '" +
                              chooser.getSelectedFile().getPath());
      } else
      {  System.out.println("No file selected.");
      }
   }
}
```



9.8.2: Displaying Images from a File (1/2)

```
import javax.swing.*;
import java.io.File;

public class Main extends Object
{  public static void main(String[ ] args)
   {  // create the dialog box and set up filters for what it displays
      JFileChooser chooser = new JFileChooser();

      // show the dialog
      int result = chooser.showOpenDialog(null);
      if (result == JFileChooser.APPROVE_OPTION)
      {  JComponent imageComp =
                  new ImageComponent(chooser.getSelectedFile().getPath());

         JFrame f = new JFrame("Image");
         f.setContentPane(imageComp);
         f.setSize(500, 500);
         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         f.setVisible(true);
      } else
      {  System.out.println("No file selected.");
      }
   }
}
```

9.8.2: Displaying Images from a File (2/2)

```
import java.awt.*;
import javax.swing.*;

/** A component that paints an image stored in a file.
*
* @author Byron Weber Becker */
public class ImageComponent extends JComponent
{
   private ImageIcon image;

   /** Construct the new component.
   * @param fileName The file where the image is stored. */
   public ImageComponent(String fileName)
   {  super();
      this.image = new ImageIcon(fileName);
      this.setPreferredSize(new Dimension(this.image.getIconWidth(),
                                          this.image.getIconHeight()));
   }

   /** Paint this component, including its image. */
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      g.drawImage(this.image.getImage(), 0, 0, null);
   }
}
```

9.9.1: The Open File for Input Pattern

Name: Open File for Input

Context: You need to read information stored in a file.

Solution: Open the file and use **Scanner** to obtain the individual tokens within the file. The following template applies:

```
Scanner «in» = null;
try
{  «in» = new Scanner(new File(«fileName»));
} catch (FileNotFoundException ex)
{  System.out.println(ex.getMessage());
   System.out.println("in " + System.getProperty("user.dir"));
   System.exit(-1);
}
«statements to read file»
«in».close();
```

Consequences: The file is opened for reading. If it does not exist, an exception is thrown.

Related Patterns: Open File for Output is used to write information to a file. Process File depends on this pattern to open the file.

9.9.2: The Open File for Output Pattern

Name: Open File for Output

Context: You need to write information to a file.

Solution: Open the file for output using the **PrintWriter** class:

```
PrintWriter «out» = null;
try
{ «out» = new PrintWriter(«fileName»);
} catch (FileNotFoundException ex)
{ System.out.println(ex.getMessage());
  System.out.println("in " + System.getProperty("user.dir"));
  System.exit(-1);
}
«statements to output to file»
«out».close();
```

Consequences: The file is opened for writing. An exception is thrown if the file cannot be created.

Related Patterns: This pattern is similar to Open File for Input.

9.9.3: The Process File Pattern

Name: Process File

Context: You need to process all of the records in a data file, one after another.

Solution: Use an instance of **Scanner** to provide the records, one at a time. A **while** loop that tests for the end of the file controls processing.

```
Scanner in = this.openFile(«fileName»);
while (in.hasNextLine())
{ «read one record»
  «process one record»
}
in.close();
```

A record is usually represented by an object. Reading is often best done in a constructor or factory method belonging to the class.

Consequences: The file is read record-by-record from beginning to end.

Related Patterns: Open File for Input is used in **openFile**. *«read one record»* is often performed using the Construct Record from File pattern.

Name: Construct Record from File

Context:

Solution:

Consequences:

Related Patterns:

Potential Homework Assignment

9.9.5: The Error-Checked Input Pattern

Name: Error-Checked Input

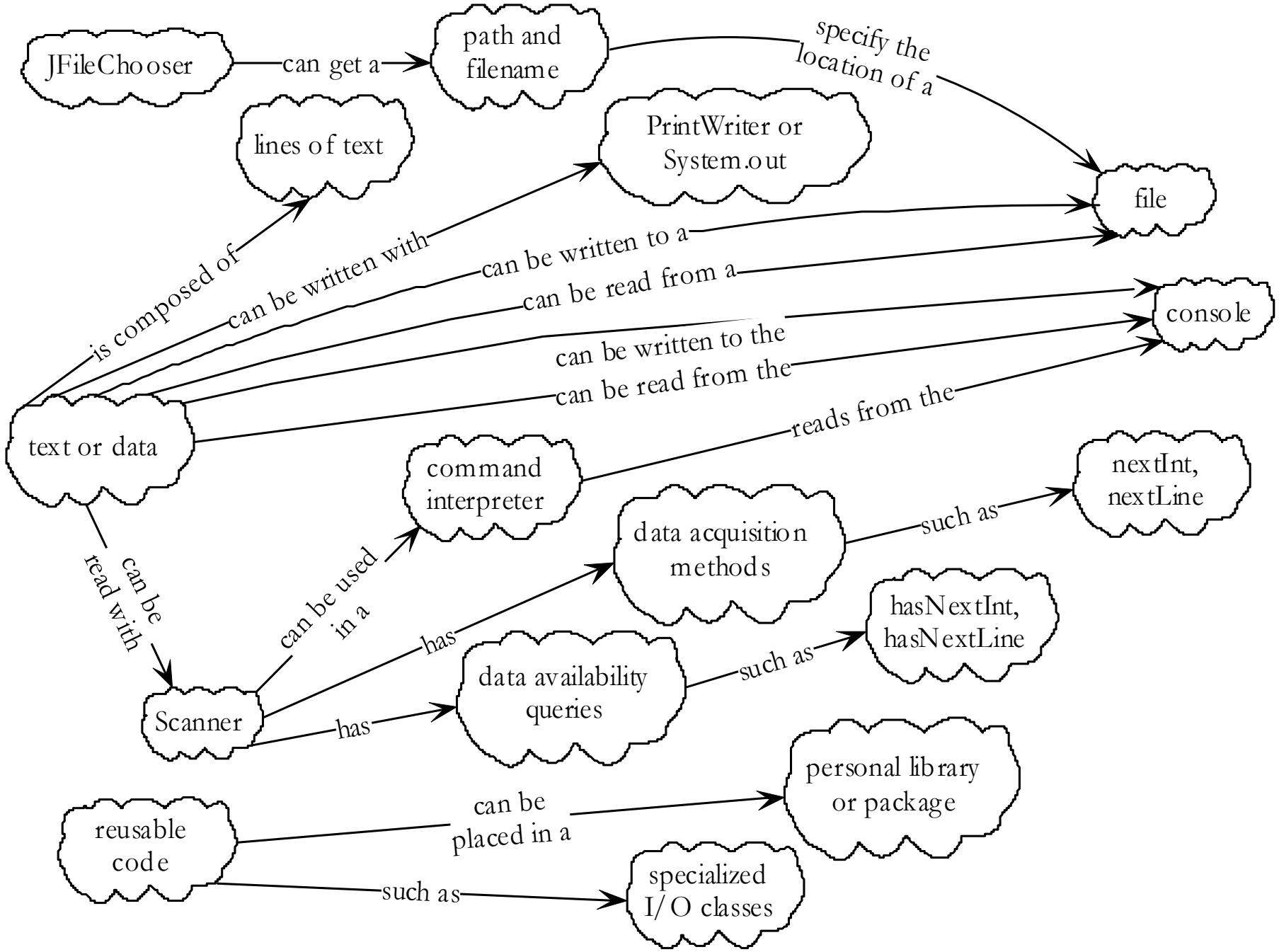
Context: Input from the user is required. Because the user may enter erroneous data, error checking is appropriate.

Solution: This pattern can be simplified if only the type of the input needs to be checked.

```
Scanner in = new Scanner(System.in);
«answerType» answer = «initialValue»;
boolean ok = false;
System.out.print(«prompt»);
while (!ok)
{  if (in.«hasNextType»())
   {  answer = in.«nextType»();
      if («testForCorrectValue»)   {  ok = true;     }
      else  {  System.out.print(«incorrectValueErrorPrompt»);  }
   } else
   {  System.out.print(«incorrectTypeErrorPrompt»);
      in.nextLine();
   }
}
```

Consequences: Program loops until user enters an expected value.

Concept Map


JFileChooser
can get a
path and filename
specify the location of a
file
lines of text
PrintWriter or System.out
is composed of
can be written with
can be written to a
can be read from a
can be written to the
can be read from the
console
text or data
reads from the
command interpreter
nextInt, nextLine
such as
data acquisition methods
can be read with
can be used in a
has
hasNextInt, hasNextLine
such as
data availability queries
has
Scanner
personal library or package
can be placed in a
reusable code
such as
specialized I/O classes

Summary

We have learned:

- how to open a file and read information from it, and write to a file.
- that files consist of a sequence of characters and that sequences of characters between delimiters are tokens.
- how to use data acquisition methods to read the next token and data availability methods to determine the type of the next token.
- how to read a record as an object.
- that file formats affect the complexity of the code that reads it.
- how to specify file locations using absolute and relative paths, and how to use the **File** class.
- how to obtain information from the user via the console and how to check it for errors.
- how to use **JFileChooser** to make it easier for users to specify files and how to display a picture stored in a file.