University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Pitt SCI 5 YEARS

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

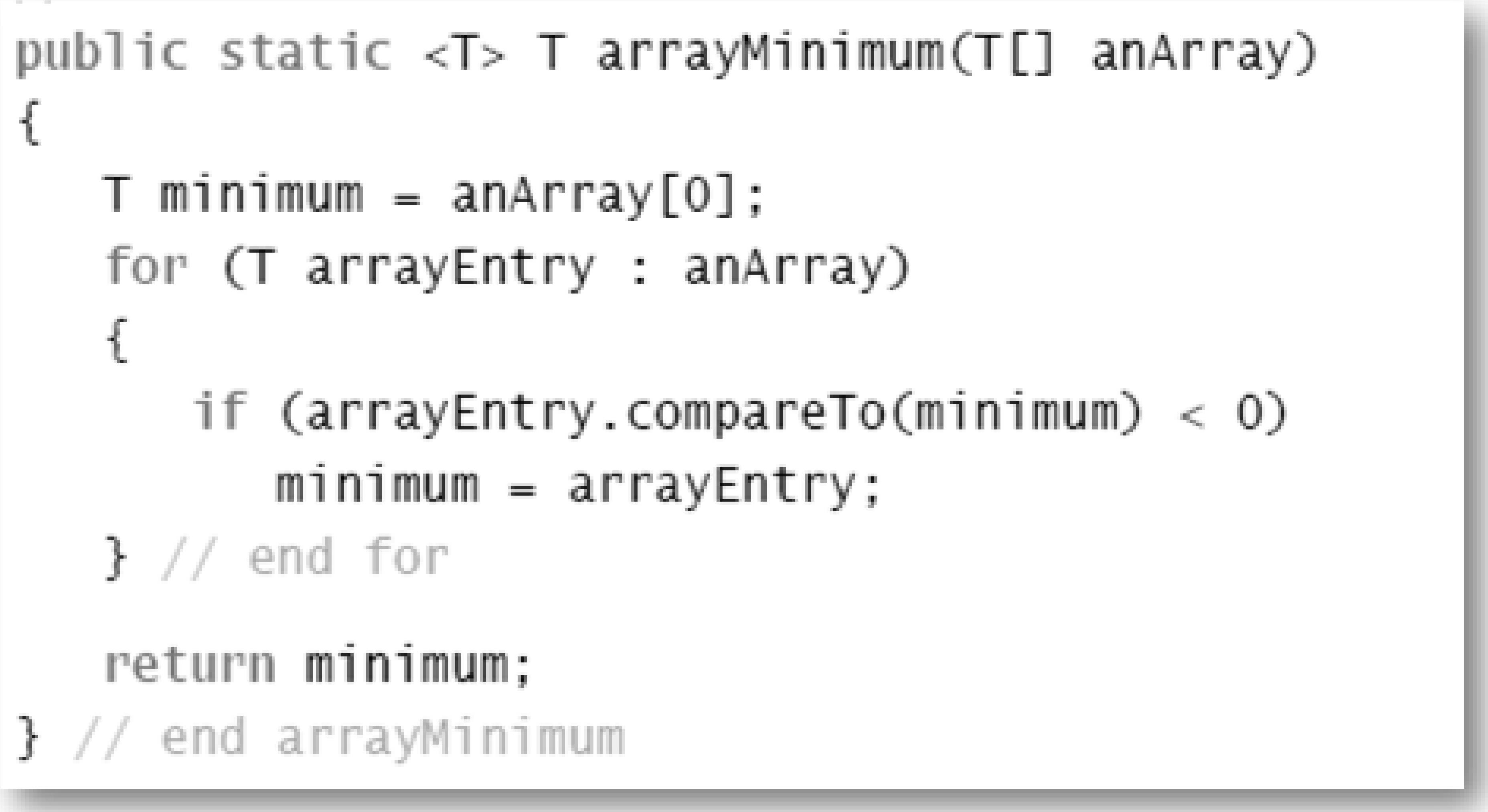
- Recitations start this week
- Homework 1 due this Friday
- Draft slides and handouts available on Canvas

Today's Agenda

- Abstract Data Types
  - Generics
- File Operations
- ArrayBag

Bounded Type Parameters

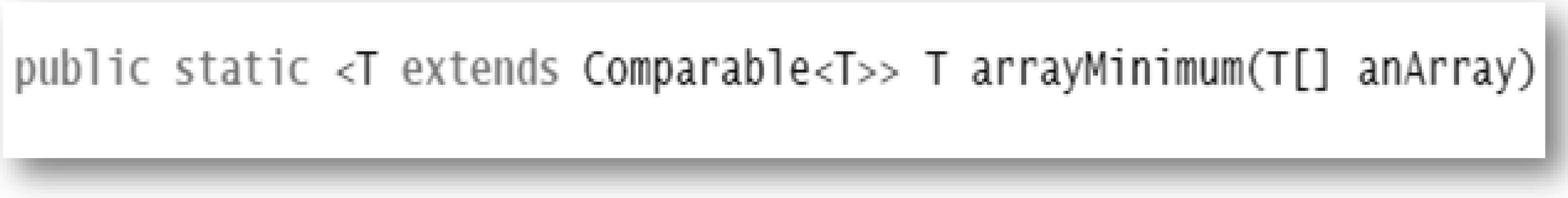
- Imagine that we want to write a static method that returns the smallest object in an array.
  Suppose that we wrote our method shown above



```
public static <T> T arrayMinimum(T[] anArray)
{
   T minimum = anArray[0];
   for (T arrayEntry : anArray)
   {
      if (arrayEntry.compareTo(minimum) < 0)
         minimum = arrayEntry;
   } // end for

   return minimum;
} // end arrayMinimum
```

Bounded Type Parameters

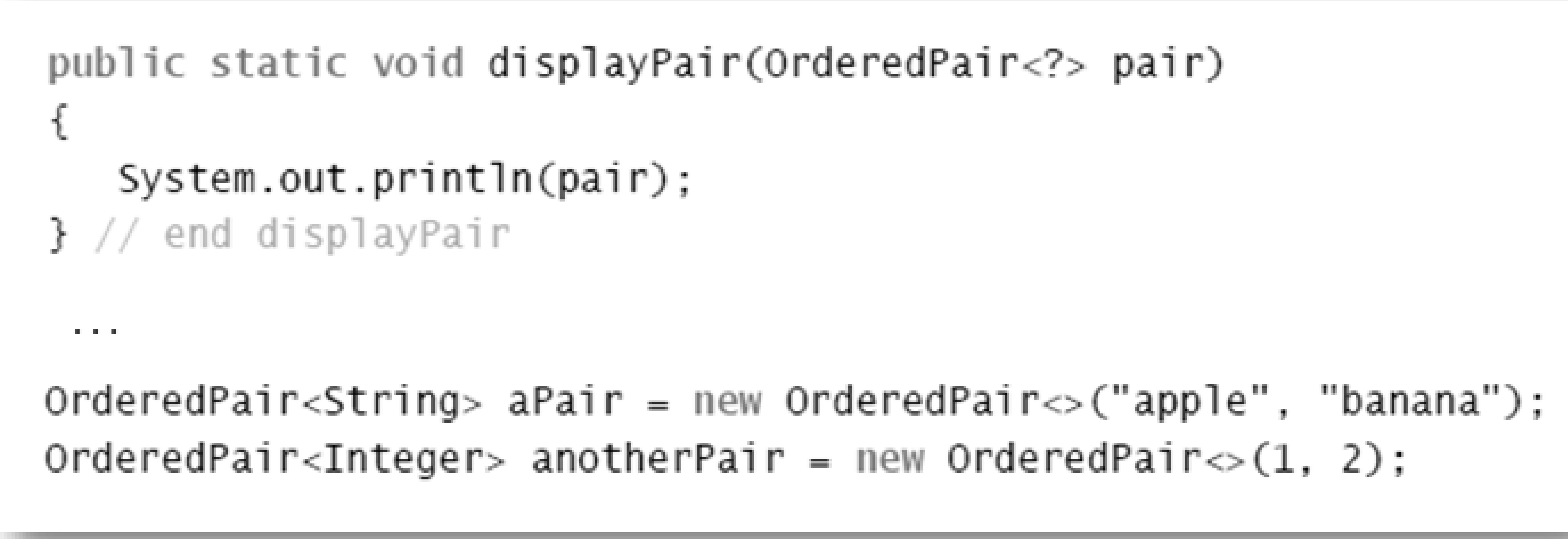
- Header really should be as shown



```
public static <T extends Comparable<T>> T arrayMinimum(T[] anArray)
```

Wildcards

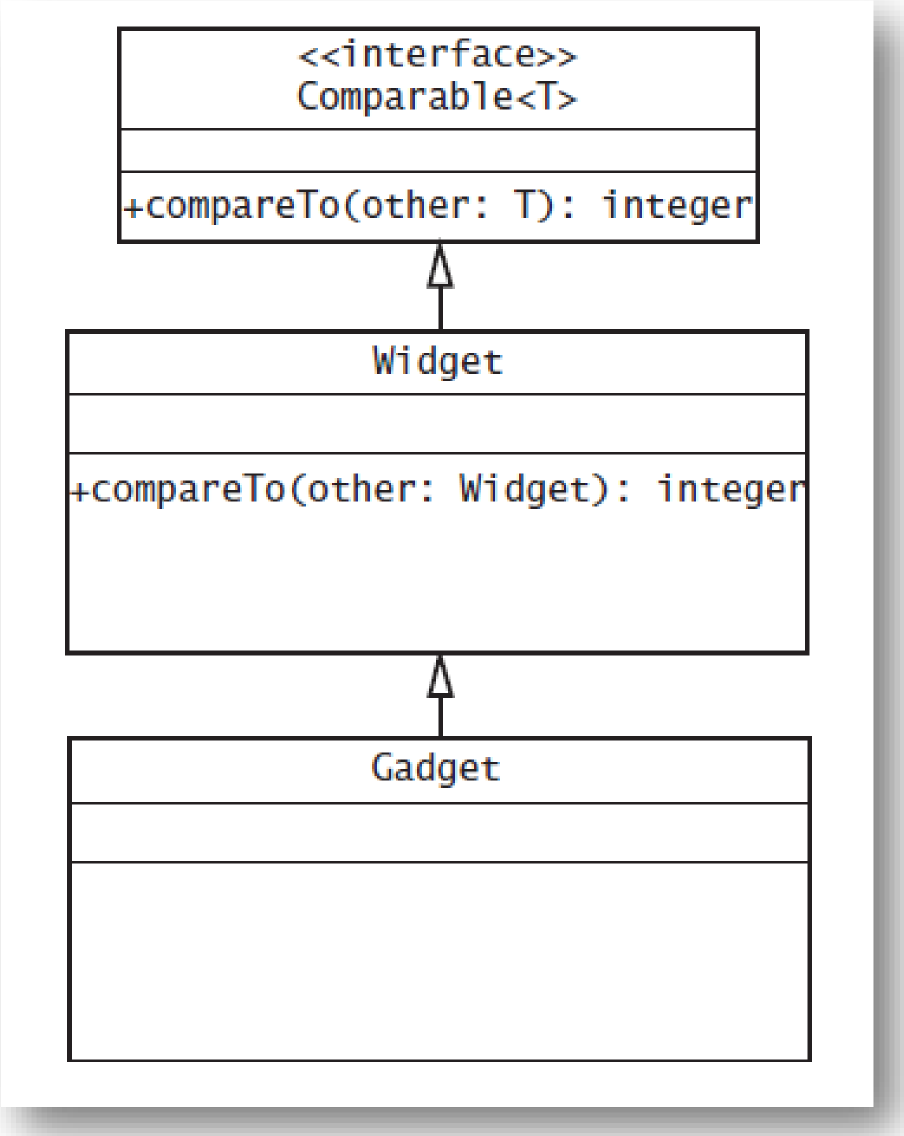
- Question mark, ?, is used to represent an unknown class type
  - Referred to as a wildcard
- Method **displayPair** will accept as an argument a pair of objects whose data type is any one class



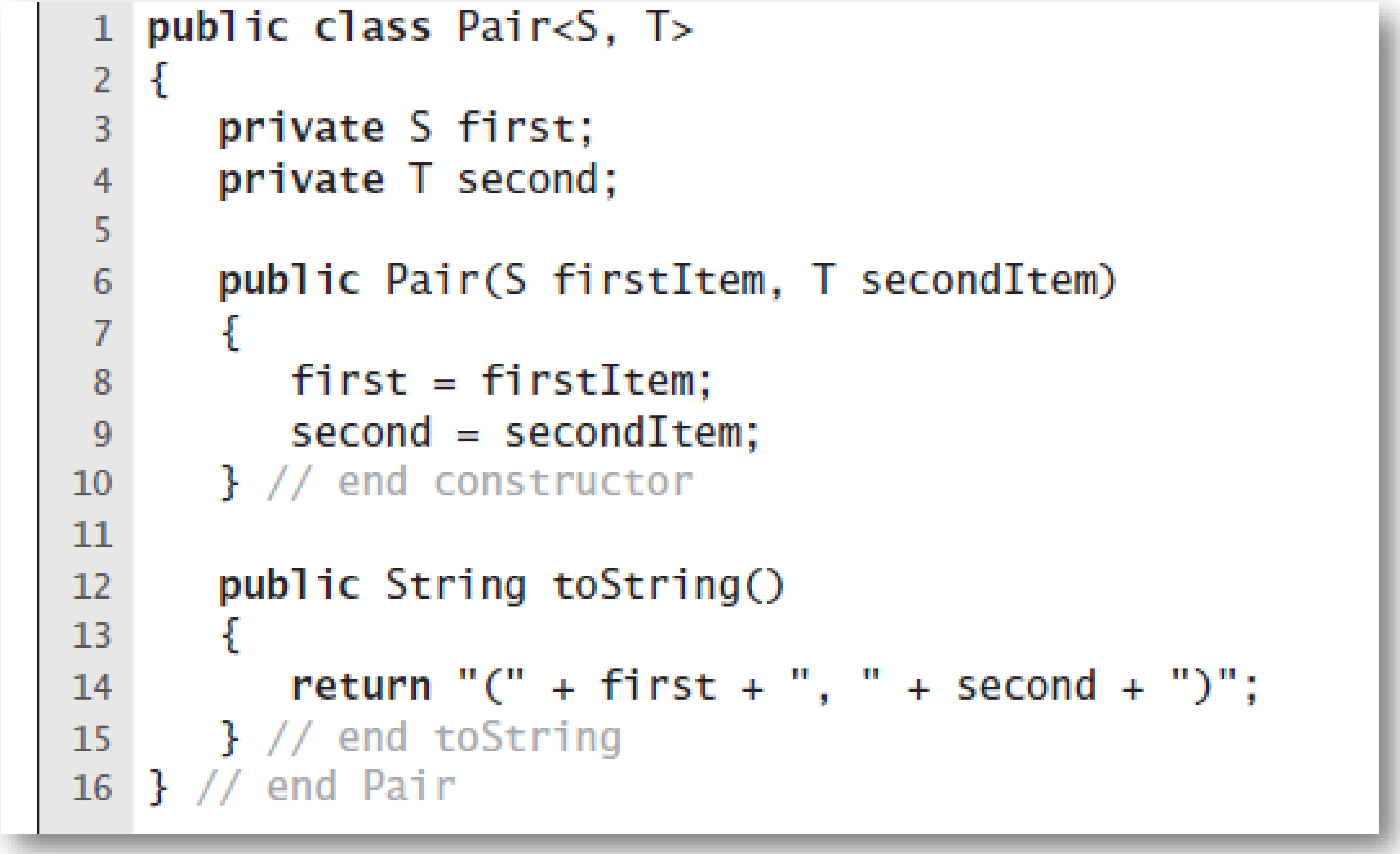
```
public static void displayPair(OrderedPair<?> pair)
{
   System.out.println(pair);
} // end displayPair

 ...

OrderedPair<String> aPair = new OrderedPair<>("apple", "banana");
OrderedPair<Integer> anotherPair = new OrderedPair<>(1, 2);
```

Bounded Wildcards

- The class **Gadget** is derived from the class **Widget**, which implements the interface **Comparable**
- **UML diargam**


<<interface>>
Comparable<T>
+compareTo(other: T): integer
Widget
+compareTo(other: Widget): integer
Gadget

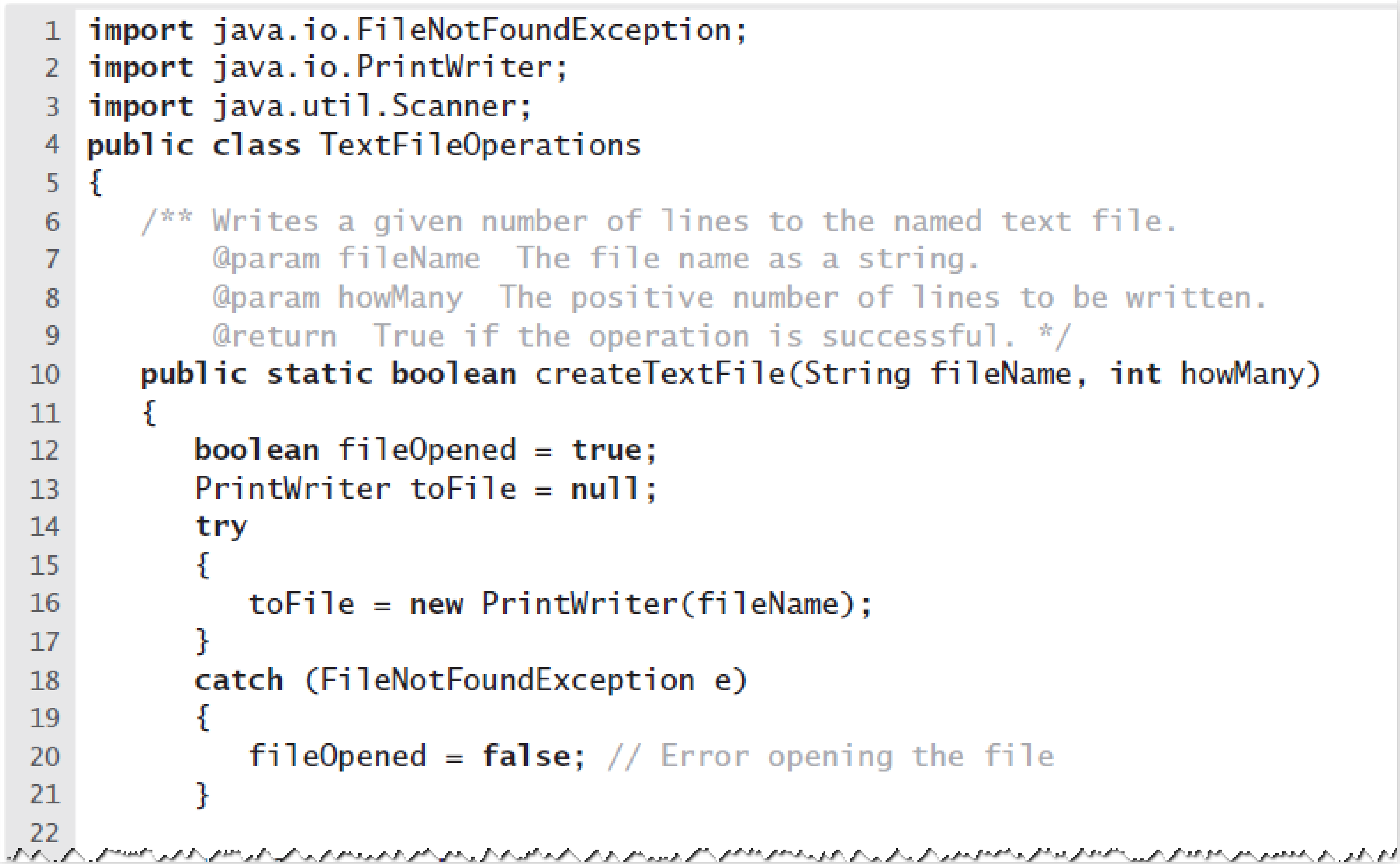
More Than One Generic Type



```
public class Pair<S, T>
{
    private S first;
    private T second;

    public Pair(S firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    } // end constructor

    public String toString()
    {
        return "(" + first + ", " + second + ")";
    } // end toString
} // end Pair
```

Writing to a Text File

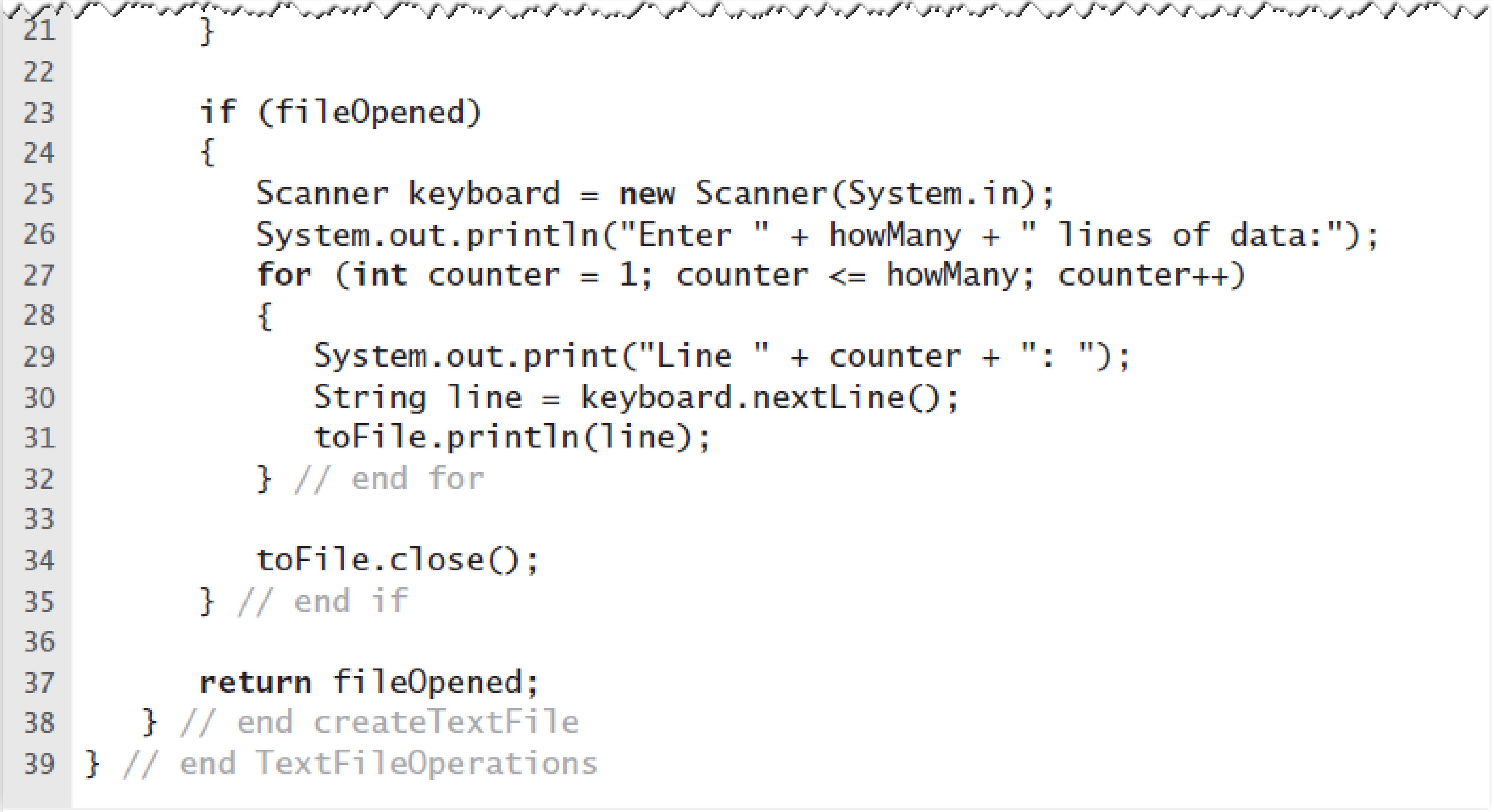
- **Using java.io.PrintWriter**



```java
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;
public class TextFileOperations
{
    /** Writes a given number of lines to the named text file.
        @param fileName  The file name as a string.
        @param howMany  The positive number of lines to be written.
        @return  True if the operation is successful. */
    public static boolean createTextFile(String fileName, int howMany)
    {
        boolean fileOpened = true;
        PrintWriter toFile = null;
        try
        {
            toFile = new PrintWriter(fileName);
        }
        catch (FileNotFoundException e)
        {
            fileOpened = false; // Error opening the file
        }

```

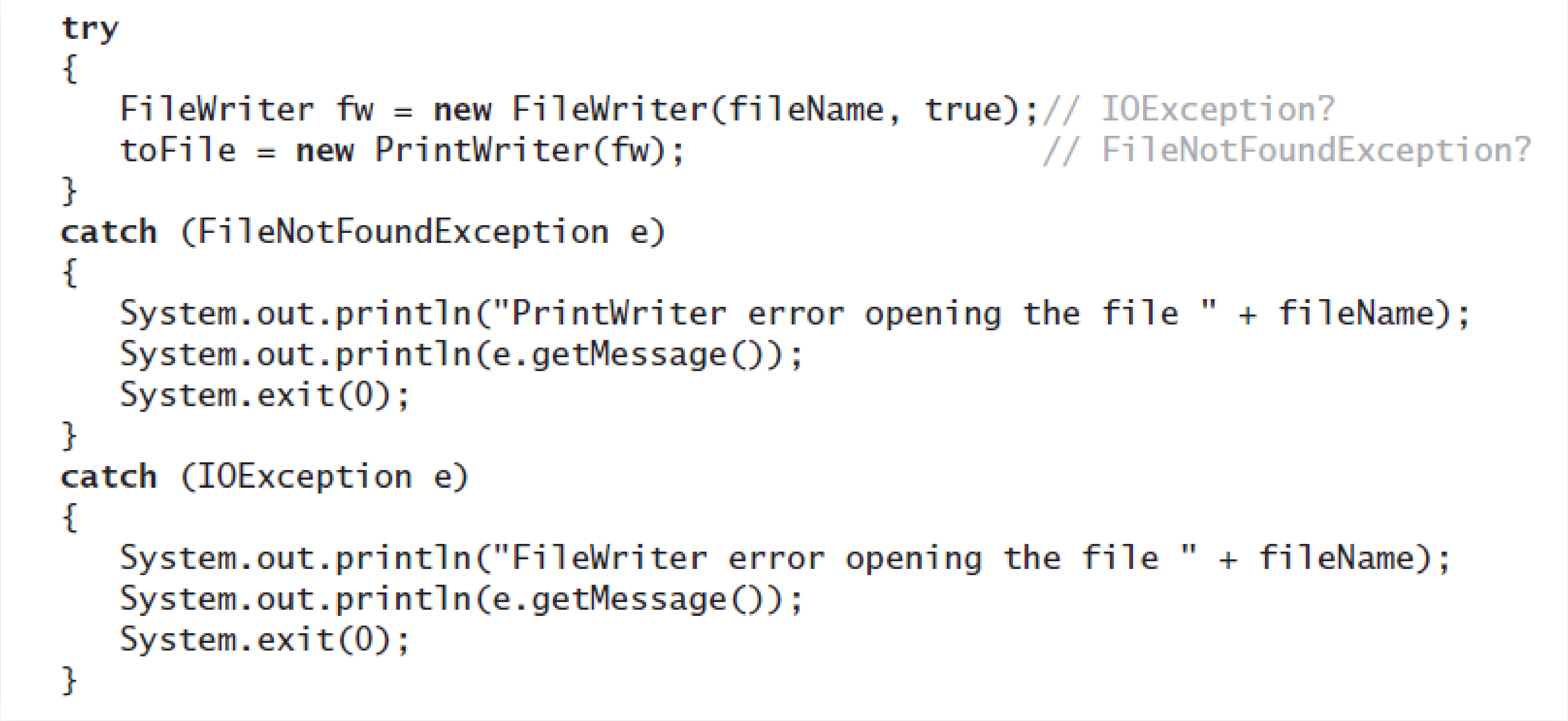
Writing to a Text File

- **Using java.io.PrintWriter.println**



```
      }

      if (fileOpened)
      {
         Scanner keyboard = new Scanner(System.in);
         System.out.println("Enter " + howMany + " lines of data:");
         for (int counter = 1; counter <= howMany; counter++)
         {
            System.out.print("Line " + counter + ": ");
            String line = keyboard.nextLine();
            toFile.println(line);
         } // end for

         toFile.close();
      } // end if

      return fileOpened;
   } // end createTextFile
} // end TextFileOperations
```

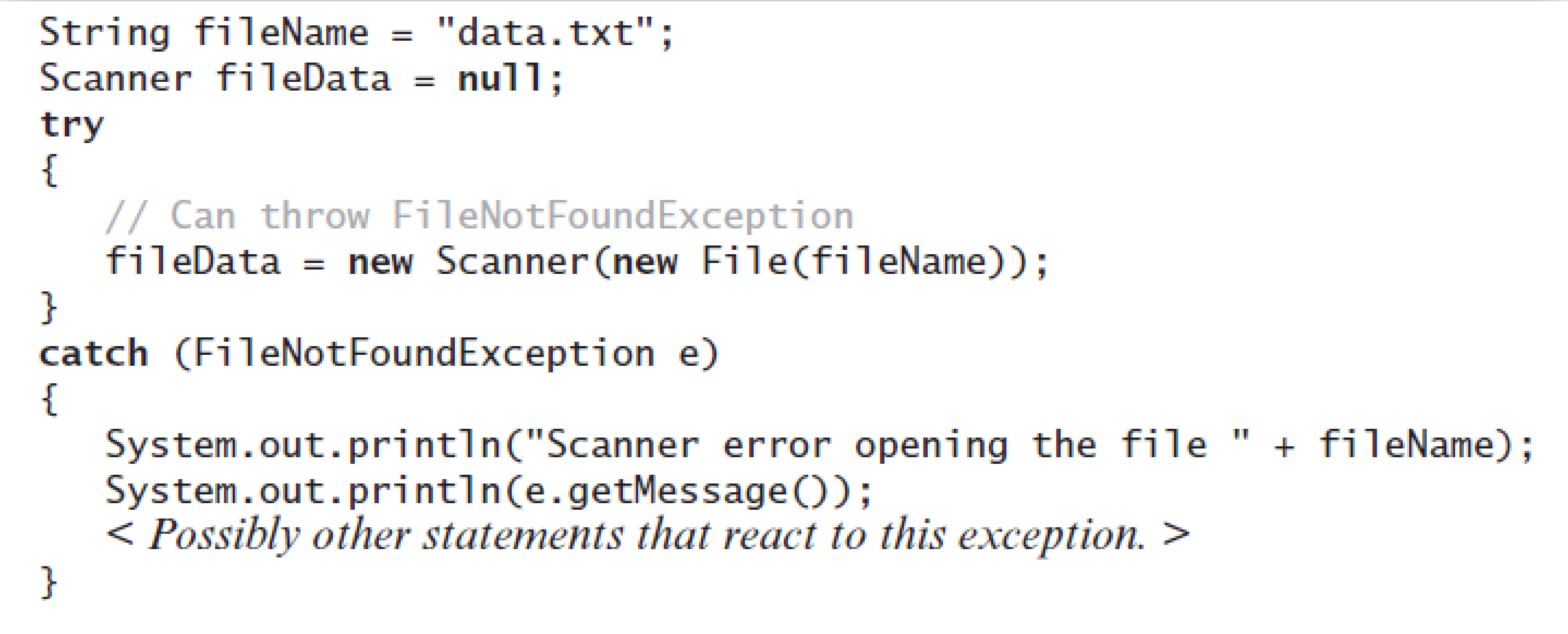
FileWriter vs. PrintWriter (Appending)



```
try
{
   FileWriter fw = new FileWriter(fileName, true);// IOException?
   toFile = new PrintWriter(fw);                  // FileNotFoundException?
}
catch (FileNotFoundException e)
{
   System.out.println("PrintWriter error opening the file " + fileName);
   System.out.println(e.getMessage());
   System.exit(0);
}
catch (IOException e)
{
   System.out.println("FileWriter error opening the file " + fileName);
   System.out.println(e.getMessage());
   System.exit(0);
}
```

Reading a Text File

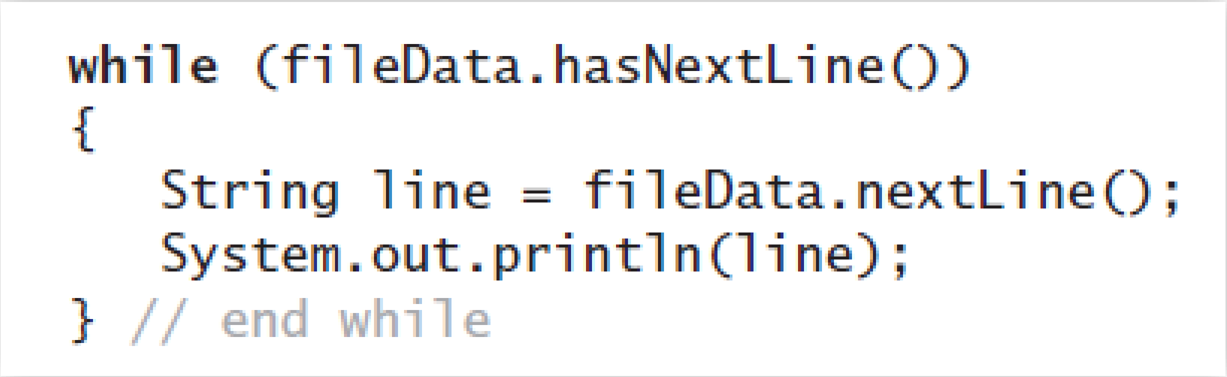
- Opening the text file named **data.txt** for input



```
String fileName = "data.txt";
Scanner fileData = null;
try
{
   // Can throw FileNotFoundException
   fileData = new Scanner(new File(fileName));
}
catch (FileNotFoundException e)
{
   System.out.println("Scanner error opening the file " + fileName);
   System.out.println(e.getMessage());
   < Possibly other statements that react to this exception. >
}
```

Reading a Text File

- If you do not know format of the data in file,
  - Use the **Scanner** method **nextLine** to read it line by line.



```
while (fileData.hasNextLine())
{
   String line = fileData.nextLine();
   System.out.println(line);
} // end while
```

Bag ADT

- The Bag
  - Think of a real bag in which we can place things
  - No rule about how many items to put in
  - No rule about the order of the items
  - No rule about duplicate items
  - No rule about what type of items to put in
    - However, we will make it homogeneous by requiring the items to be the same class or subclass of a specific Java type
  - Let's look at the interface
    - See BagInterface.java

ADT Bag

- Note what is NOT in the interface:
  - Any specification of the data for the collection
    - We will leave this to the implementation
    - The interface specifies the behaviors only
      - However, the implementation is at least partially implied
      - Must be some type of collection
  - Any implementation of the methods
- Note that other things are not explicitly in the interface but maybe should be
  - Ex: What the method should do
  - Ex: How special cases should be handled
  - We typically have to handle these via comments

ADT Bag

- Ex: `public boolean add(T newEntry)`
  - We want to consider specifications from two points of view:
    1) What is the purpose / effect of the operation in the normal case?
    2) What unusual / erroneous situations can occur and how do we handle them?
  - The first point can be handled via preconditions and postconditions
    - Preconditions indicate what is assumed to be the state of the ADT prior to the method's execution
    - Postconditions indicate what is the state of the ADT after the method's execution
    - From the two we can infer the method's effect

ADT Bag

- Ex: for add(newEntry) we might have:

  Precondition:

  Bag is in a valid state containing N items

  Postconditions:

  Bag is in a valid state containing N+1 items

  newEntry is now contained in the Bag

- This is somewhat mathematical, so many ADTs also have operation descriptions explaining the operation in plainer terms
    - More complex operations may also have more complex conditions
  - However, pre and postconditions can be very important for verifying correctness of methods

ADT Bag

- The second point (abnormal cases) is often trickier to handle
  - Sometimes the unusual / erroneous circumstances are not obvious
  - Often they can be handled in more than one way
  - Ex: for add(newEntry) we might have
    - Bag is not valid to begin with due to a previous error
    - newEntry is not a valid object
  - Assuming we detect the problem, we could handle it by
    - Doing a "no op"
    - Returning a false boolean value
    - Throwing an exception
  - We need to make these clear to the user of the ADT so they know what to expect