Static Declarations

- Storage allocation for static objects, fields, methods.
- 2 ways to invoke static fields & methods: using an object or using the class name.

Arrays

- Reference types; explicitly created using new statement.
- Index starts at 0.
- Arrays have length field.
- Array assignment ≠ array copy.
- Array copy done using clone()
- multi-dimensional arrays
- Dynamic arrays – automatic using ArrayList

Figure 2.6, 2.7, page 42-43

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

public class ReadStrings
{
    public static void main( String [ ] args )
    {
        String [ ] array = getStrings( );
        for( int i = 0; i < array.length; i++ )
            System.out.println( array[ i ] );
    }

    // Read an unlimited number of String; return a String [ ]
    public static String [ ] getStrings( )
    {
        BufferedReader in = new BufferedReader( new InputStreamReader( System.in ) );
        String [ ] array = new String[ 5 ];
        int itemsRead = 0;
        String oneLine;

        System.out.println( "Enter any number of strings, one per line; " );
        System.out.println( "Terminate with empty line: " );

        try
        {
            while( ( oneLine = in.readLine( ) ) != null && !oneLine.equals( "" ) )
            {
                if( itemsRead == array.length )
                    array = resize( array, array.length * 2 );
                array[ itemsRead++ ] = oneLine;
            }
        }
        catch( IOException e )
        {
            System.out.println( "Unexpected IO Exception has shortened amount read" );
        }

        System.out.println( "Done reading" );
        return resize( array, itemsRead );
    }
```

Figure 2.7, page 43

```
// Resize a String[ ] array; return new array
    public static String [ ] resize( String [ ] array, int newSize )
    {
        String [ ] original = array;
        int numToCopy = Math.min( original.length, newSize );

        array = new String[ newSize ];
        for( int i = 0; i < numToCopy; i++ )
            array[ i ] = original[ i ];
        return array;
    }
}
```

Figure 2.8, page 44

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.ArrayList;
public class ReadStringsWithArrayList
{
    public static void main( String [ ] args )
    {
        ArrayList array = getStrings( );
        for( int i = 0; i < array.size( ); i++ )
            System.out.println( array.get( i ) );
    }

    // Read an unlimited number of String; return an ArrayList
    public static ArrayList getStrings( )
    {
        BufferedReader in = new BufferedReader( new InputStreamReader( System.in ) );
        ArrayList array = new ArrayList( );
        String oneLine;

        System.out.println( "Enter any number of strings, one per line; " );
        System.out.println( "Terminate with empty line: " );

        try
        {
            while( ( oneLine = in.readLine( ) ) != null && !oneLine.equals( "" ) )
                array.add( oneLine );
        }
        catch( IOException e )
        {
            System.out.println( "Unexpected IO Exception has shortened amount read" );
        }

        System.out.println( "Done reading" );
        return array;
    }
}
```

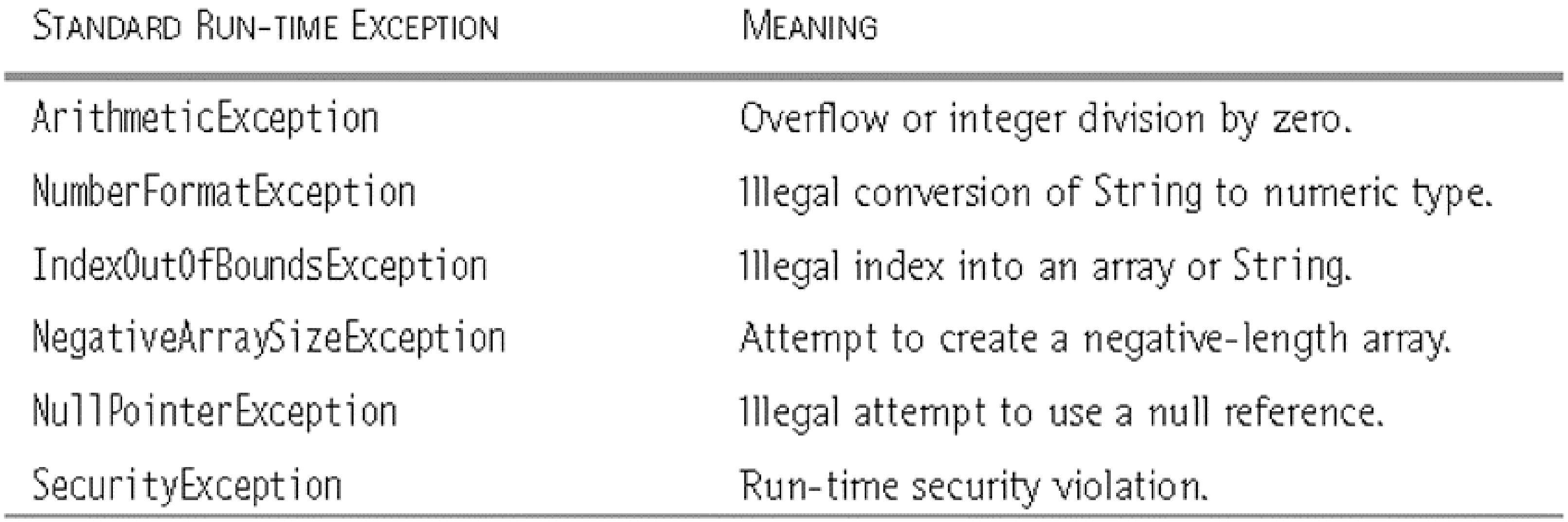
Exceptions & Errors

- An exception is an object that is thrown from the site of an error and can be caught by an appropriate exception handler.
- Separating the handler from error detection makes the code easier to read and write. Do not use exception as a “cheap” goto statement. Better to pass it on to calling procedure.
- More reliable error recovery without simply exiting.
- User-defined exceptions can be created or thrown.
- The try region is a guarded region from which errors can be caught by exceptions.

- Errors are virtual machine problems. OutOfMemoryError, InternalError, UnknownError are examples of errors.
- Errors are uncrecoverable and should not be caught.

Figure 2.12

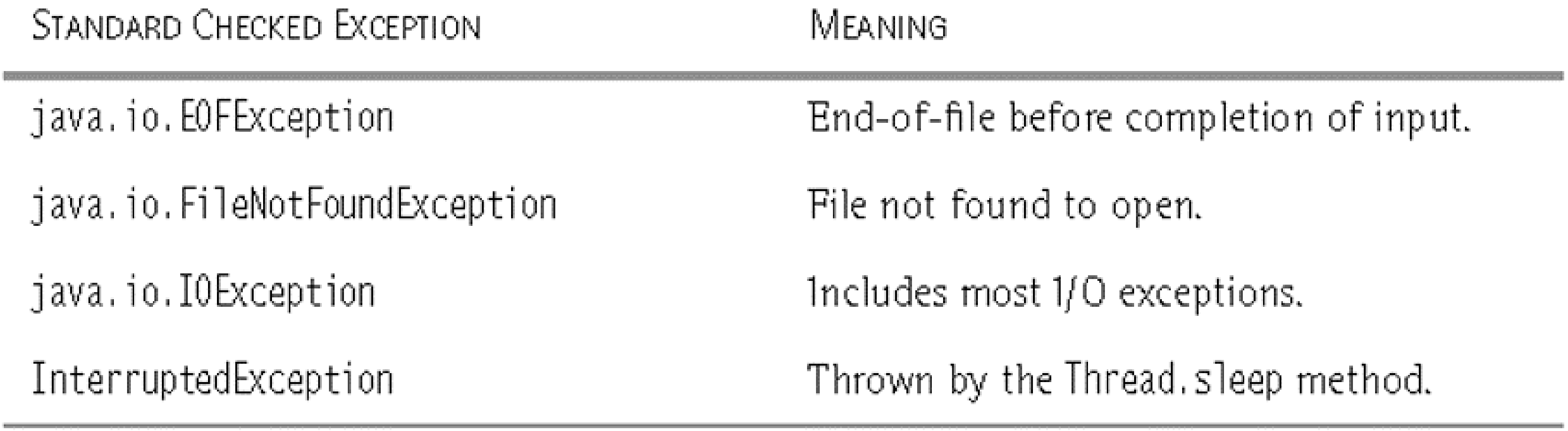
Common standard run-time exceptions



Standard Run-time Exception	Meaning
ArithmeticException	Overflow or integer division by zero.
NumberFormatException	Illegal conversion of String to numeric type.
IndexOutOfBoundsException	Illegal index into an array or String.
NegativeArraySizeException	Attempt to create a negative-length array.
NullPointerException	Illegal attempt to use a null reference.
SecurityException	Run-time security violation.

Figure 2.13

Common standard checked exceptions



STANDARD CHECKED EXCEPTION	MEANING
`java.io.EOFException`	End-of-file before completion of input.
`java.io.FileNotFoundException`	File not found to open.
`java.io.IOException`	Includes most I/O exceptions.
`InterruptedException`	Thrown by the `Thread.sleep` method.

Input/Output

- Streams are used for I/O
- Terminal I/O treated in the same way as File I/O.
- Predefined streams System.in, System.out, System.err
- readLine and StringTokenizer are useful methods for formatted input; they are part of java.util.StringTokenizer

Figure 2.15, page 53

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.StringTokenizer;
public class MaxTest
{
    public static void main( String [ ] args )
    {
        BufferedReader in = new BufferedReader( new InputStreamReader( System.in ) );
        String oneLine;
        StringTokenizer str;
        int x, y;

        System.out.println( "Enter 2 ints on one line: " );
        try
        {
            oneLine = in.readLine( );
            if( oneLine == null )
                return;

            str = new StringTokenizer( oneLine );
            if( str.countTokens( ) != 2 )
            {
                System.out.println( "Error: need two ints" );
                return;
            }
            x = Integer.parseInt( str.nextToken( ) );
            y = Integer.parseInt( str.nextToken( ) );
            System.out.println( "Max: " + Math.max( x, y ) );
        }
        catch( IOException e )
          { System.err.println( "Unexpected IO error" ); }
        catch( NumberFormatException e )
          { System.err.println( "Error: need two ints" ); }
    }
}
```

Figure 2.16, page 54

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

public class ListFileContents
{
   public static void main( String [ ] args )
   {
      if( args.length == 0 ) System.out.println( "No files specified" );
      for( int i = 0; i < args.length; i++ )  listFile( args[ i ] );
   }
   public static void listFile( String fileName )
   {
      FileReader theFile;
      BufferedReader fileIn = null;
      String oneLine;
      System.out.println( "FILE: " + fileName );
      try
      {
         theFile = new FileReader( fileName );
         fileIn  = new BufferedReader( theFile );
         while( ( oneLine = fileIn.readLine( ) ) != null )
            System.out.println( oneLine );
      }
      catch( IOException e )
        {  System.out.println( e ); }
      finally
      {
         // Close the stream
         try
         {
            if (fileIn != null )  fileIn.close( );
         }
         catch( IOException e ) { }
      }
   }
}
```

Figure 2.17, page 56

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
public class DoubleSpace
{
   public static void main( String [ ] args )
   {
      for( int i = 0; i < args.length; i++ )
         doubleSpace( args[ i ] );
   }
   public static void doubleSpace( String fileName )
   {
      PrintWriter   fileOut = null;
      BufferedReader fileIn = null;
      try
      {
         fileIn  = new BufferedReader( new FileReader( fileName ) );
         fileOut = new PrintWriter(  new FileWriter( fileName + ".ds" ) );
         String oneLine;
         while( ( oneLine = fileIn.readLine( ) ) != null )
            fileOut.println( oneLine + "\n" );
      }
      catch( IOException e )     { e.printStackTrace( ); }
      finally
      {
         try
         {
            if( fileOut != null )     fileOut.close( );
            if( fileIn != null )      fileIn.close( );
         }
         catch( IOException e )
           { e.printStackTrace( ); }
      }
   }
}
```

Objects & Classes

- Difference between class and object
- Private, public, protected, package visibility
- Basic methods:

  constructors, mutators, assessors, output, equals.
- Expression to check type of object: instanceof.
- Reference to current object & constructor: this.
- Global constant: static final

Packages

- Group of related classes.
- Specified by package statement.
- Fewer restrictions on access among each other;
  - if class is called public, then it is visible to all classes
  - if no visibility modifier is specified, its visibility is termed as “package visibility” and is somewhere between:
    - private (other classes in package cannot access it) and
    - public (other classes outside package can also access it)
- Package locations can be specified by environmental variables.

Defining a Class

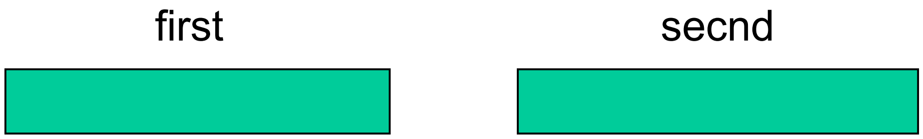
- The Student class describes a single student. It contains a single instance field named lastName
- Each Student object will have a unique copy of its own instance fields

```
public class Student {
  String lastName;
}
```

Declaring an Object

- The Student class describes a single student. It contains a single instance field named lastName
- Each Student object contains a distinct copy of its instance fields

```
Student first = new Student();
Student secnd = new Student();
```



Add a Constructor

- Executed when an object is created
- Same name as the class
- No return type
- Without parameters, it is called a default constructor

```
public class Student {
  public Student()
  {
    lastName = "(none)";
  }

  String lastName;
}
```

Add a toString Method

- The toString() method is already defined in the Object class
- We can provide our own version here

```
class Student {

  Student()
  {
    lastName = "Smith";
  }

  public String toString()
  {
    return "Last name = " + lastName;
  }

  String lastName;
}
```

Add a Public Test Class

- Every program must have a public class that contains main()
- Keep this class short and simple

```
public class StudentTest {

  public static void main( String args[] )
  {
    Student S = new Student();
    System.out.println( S.toString() );
  }
}

// (See the Student1 project)
```

Add a Second Constructor

- This constructor has a String parameter that initializes the lastName instance field

```
public class Student {

   public Student( String aName )
   {
      lastName = aName;
   }

}
```

Selectors and Mutators

- A selector method returns the value of an instance field
- A mutator method changes the value of an instance field

```
public String getLastName()
{
  return lastName;
}

public void setLastName( String aName )
{
  lastName = aName;
}
```

Selectors and Mutators

- A selector method returns the value of an instance field
- A mutator method changes the value of an instance field

```
Student S2 = new Student("Ramakrishnan");

S2.setLastName("Chong");

System.out.println( "New name of S2: "
       + S2.getLastName() );
```

Using the JavaDoc Utility

- JavaDoc generates HTML documentation for your public classes and methods
- Use the /** delimiter to begin a comment, and */ to end
- Appears before classes and methods

```
/**
   A class that holds information about a single
   college student. Demonstrates an overloaded
   constructor.
*/

public class Student {
. . .
```

Using the JavaDoc Utility - 2

- Run JavaDoc from the Tools menu in JCreator
- To install JavaDoc: follow instructions on my Samples page.

```
/**
   Program entry point; creates two students
   with different names.
*/

public static void main( String args[] )
{
   . . .
```

Using the JavaDoc Utility - 3

- @param – Identifies a method parameter
- @return – describes the function return value.

```
/**
   Constructor with one parameter that sets the last name.
   @param aName a new last name which is assigned to the
   student.
*/
  public Student(String aName)
  {
    lastName = aName;
  }


// return value example:
@return a string containing the student's last name.
```

Using the Javadoc utility - 4

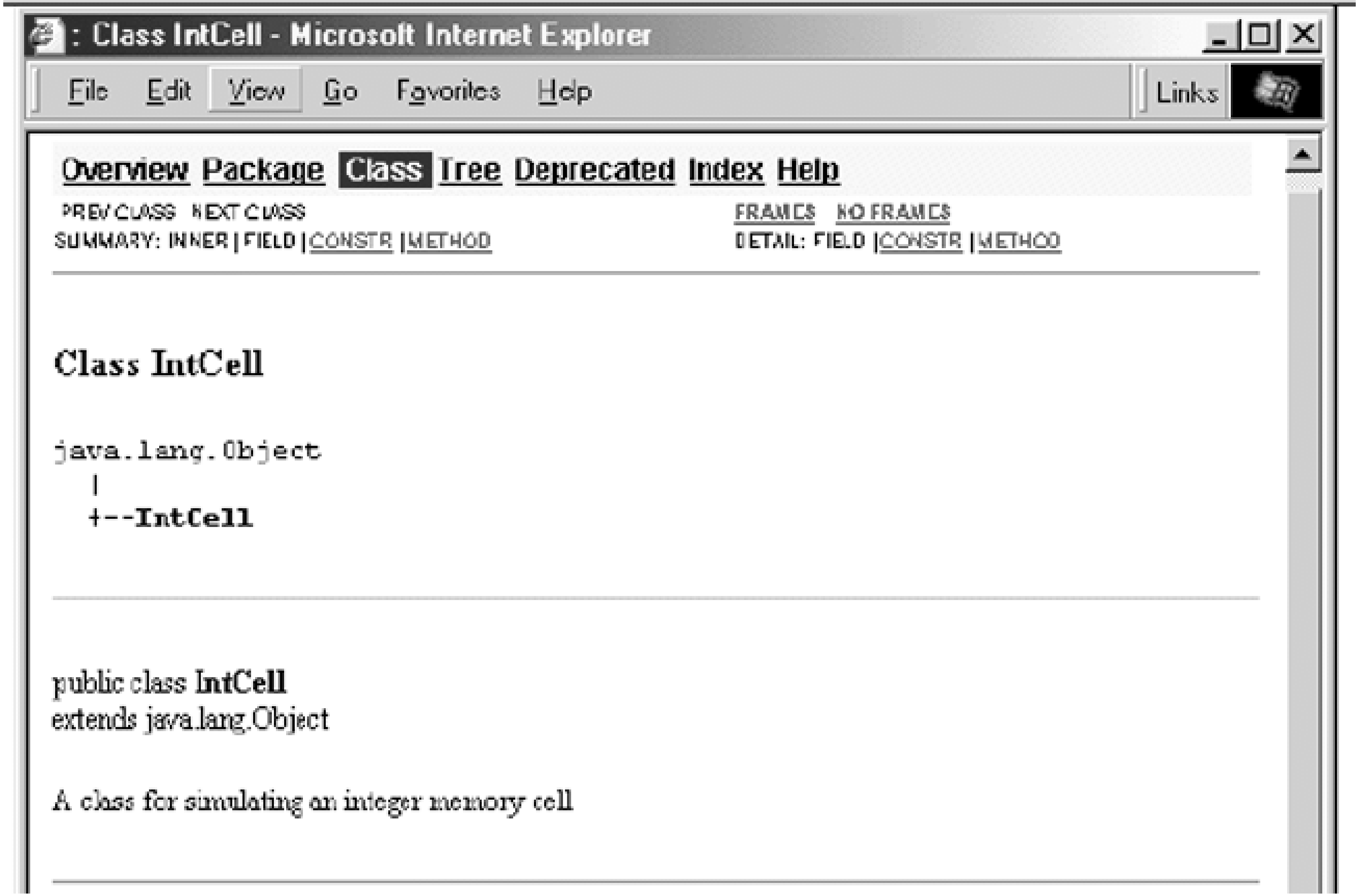
```
/**
 * A class for simulating an integer memory cell
 * @author Mark A. Weiss
 */
public class IntCell
{   /** Get the stored value.
     * @return the stored value.
     */
    public int read( )   {
        return storedValue;
    }
    /** Store a value
     * @param x the number to store.
     */
    public void write( int x )  {
        storedValue = x;
    }

    private int storedValue;
}
```

Figure 3.4, page 66

Figure 3.5 (A)

javadoc output for Figure 3.4 (partial output) (*continued*)


: Class IntCell - Microsoft Internet Explorer
File
Edit
View
Go
Favorites
Help
Links
Overview
Package
Class
Tree
Deprecated
Index
Help
PREV CLASS
NEXT CLASS
FRAMES
NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

Class IntCell

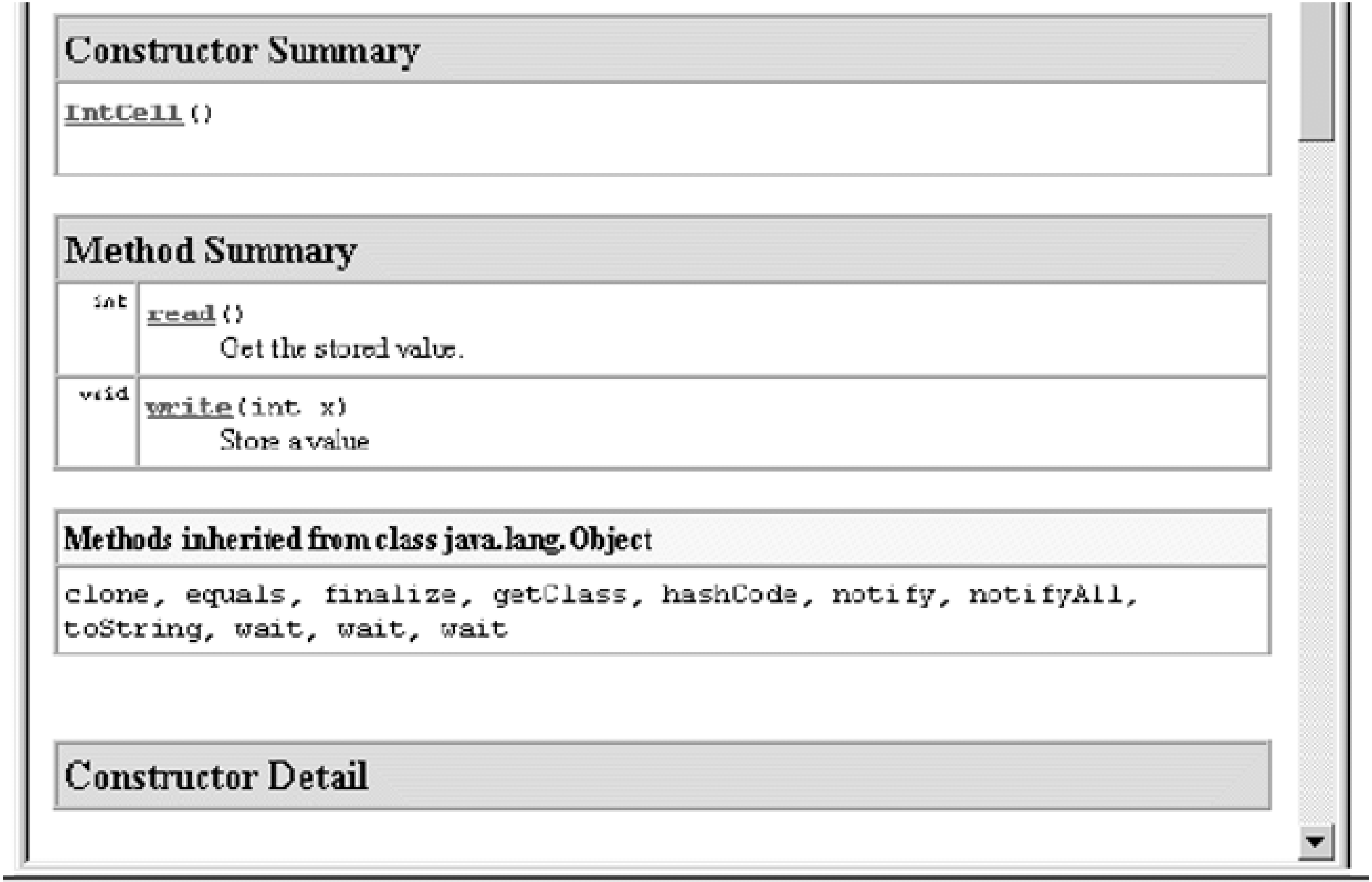
```
java.lang.Object
  |
  +--IntCell
```

public class **IntCell**
extends java.lang.Object

A class for simulating an integer memory cell

Figure 3.5 (B)

javadoc output for Figure 3.4 (partial output)



Constructor Summary

`IntCell()`

Method Summary

int	`read()` Get the stored value.
void	`write(int x)` Store a value

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Constructor Detail

Inheritance

- Defines a IS-A relationship between classes.
- Base classes and derived classes.
- Derived class inherits all fields and methods of base class.
- Derived class objects are type compatible with base class.
- protected fields and methods: visible to derived classes and to classes in same package.
- inheritance is transitive.
- polymorphism allows for redefining fields and methods.
- dynamic binding allows for run-time determination of overloads and/or overrides.
- super() is a way to refer to constructor of base class.
  It can also be called using appropriate parameters.
  It can only be the first line of a constructor.
- super with appropriate parameters is also used to invoke the corresponding method of the base class.

```
class Person // Fig 4.1, page 91
{
   public Person( String n, int ag, String ad, String p )
   {  name = n; age = ag; address = ad; phone = p;  }

   public String toString( )
   {return getName( ) + " " + getAge( ) + " " + getPhoneNumber( );  }

   public final String getName( )
   {   return name;  }

   public final int getAge( )
   {  return age;  }

   public final String getAddress( )
   {  return address;  }

   public final String getPhoneNumber( )
   {  return phone;  }

   public final void setAddress( String newAddress )
   {  address = newAddress;  }

   public final void setPhoneNumber( String newPhone )
   {  phone = newPhone; }

   private String name;
   private int age;
   private String address;
   private String phone;
}
```

```
class Student extends Person // Fig 4.8, page 102
{
   public Student( String n, int ag, String ad, String p, double g )
   {
      super( n, ag, ad, p );
      gpa = g;
   }

   public String toString( )
   {
      return super.toString( ) + " " + getGPA();
   }

   public double getGPA( )
   {
      return gpa;
   }

   private double gpa;
}
```

```
class PersonDemo // Fig 4.9, pg 103
{
    public static void printAll( Person[ ] arr )
    {
        for( int i = 0; i < arr.length; i++ )
        {
            if( arr[ i ] != null )
            {
                System.out.print( "[" + i + "] " + arr[ i ] );
                System.out.println( );
            }
        }
    }

    public static void main( String [ ] args )
    {
        Person [ ] p = new Person[ 4 ];
        p[0] = new Person( "joe", 25, "New York", "212-555-1212" );
        p[1] = new Student( "becky", 27, "Chicago", "312-555-1212", 4.0 );
        p[3] = new Employee( "bob", 29, "Boston", "617-555-1212", 100000.0 );

        if( p[3] instanceof Employee )
            ((Employee) p[3]).raise( .04 );

        printAll( p );
    }
}
```

Abstract Methods & Classes

- abstract methods are not implemented (not even a default one).
- This is better than putting in a dummy procedure as a placeholder.
- Derived classes must eventually implement them;
  if they don't then they must be abstract classes themselves.
- Overriding is resolved at runtime.
- Abstract class is one that contains an abstract method;
  need to be explicitly declared as such.
- Abstract classes may have non-abstract methods & static fields.
- Abstract classes cannot be created (no constructor),
  except using `super()`

```
public abstract class Shape
{
   public abstract double area( );
   public abstract double perimeter( );

   public double semiperimeter( )
   {  return perimeter( ) / 2; }
}
```

```
class ShapeDemo // Fig 4.11 & 4.12, pg 104-5
{
   public static double totalArea( Shape [ ] arr )
   {
      double total = 0;

      for( int i = 0; i < arr.length; i++ )
      {
         if( arr[ i ] != null )
            total += arr[ i ].area( );
      }

      return total;
   }

public static void printAll( Shape [ ] arr )
   {
      for( int i = 0; i < arr.length; i++ )
         System.out.println( arr[ i ] );
   }

   public static void main( String [ ] args )
   {
      Shape [ ] a = { new Circle( 2.0 ), new Rectangle( 1.0, 3.0 ),
                     null, new Square( 2.0 ) };
      System.out.println( "Total area = " + totalArea( a ) );
      System.out.println( "Total semiperimeter = " +
                  totalSemiperimeter( a ) );
      printAll( a );
   }
}
```