

Advanced Programming with Java

The Java™ programming platform contains a set of class libraries. These standard libraries provide functionality that ranges from file compression to graphics. Here we will discuss some of Java's fundamental classes, useful utility classes, and tools for performing I/O.

The Standard Libraries

- `java.lang`
- `java.io`
- `java.util`

Copyright ©2000-2023 by David M. Whitlock. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from whitlock@cs.pdx.edu.* Last updated January 5, 2023.

1

java.lang.Object

Object is the root class in Java: Everything is an Object

Therefore, all objects have the following methods

- `equals`: Compares an Object to another
- `toString`: Returns a String representation of an Object (often invoked automatically)
- `hashCode`: Returns a hash code for an Object
- `clone`: Returns a copy of an Object
- `getClass`: Returns an instance of an Object's "metaclass"
- `finalize`: Called when an Object is garbage collected (not a destructor!)
- `notify`, `notifyAll`, and `wait` are used in multi-threaded programs

3

The java.lang package

The `java.lang` package contains classes, interfaces, and exceptions that are fundamental to the Java programming language

- `Object`, `Class`, `System`
- `String`, `StringBuilder`
- The "wrapper" classes
- A bunch of exceptions

2

java.lang.String

Strings can be constructed from byte arrays, char arrays, or other Strings

- `charAt`: Returns the char at a given offset into a String
- `compareTo`: Compares a String to another
- `endsWith/startsWith`: Determines if one String is a suffix/prefix of another
- `indexOf`: Finds an occurrence of a char in a String
- `length`: Returns the length of a String
- `replace`: Replaces all occurrences of one char with another
- `trim`: Removes leading and trailing whitespace from a String

4

java.lang.StringBuilder

Strings are immutable, `StringBuilders`* can be changed

- `append`: Appends something to a `StringBuilder`
 - The `StringBuilder` itself is returned
 - `sb.append("Result: ").append(4);`
- `delete/deleteCharAt`: Removes some number of chars from a `StringBuilder`
- `insert`: Inserts something into a `StringBuilder`
- `length`: Returns the length of a `StringBuilder`
- `substring`: Returns a portion of a `StringBuilder` as a `String`
- `toString`: Returns the contents of a `StringBuilder` as a `String`

*`StringBuilder` provides a better-performing alternative to the older `StringBuffer` class

5

java.lang.System

Contains a number of system-level fields and methods

Static fields:

- `out`: A `PrintStream` for standard output (as in `System.out.println("Hi")`)
- `err`: A `PrintStream` for standard error
- `in`: An `InputStream` for standard in

Static methods:

- `currentTimeMillis`: Returns the current time in milliseconds since January 1, 1970
- `exit`: Shuts down the JVM with a given `int` exit code
- `setOut`, `setErr`, `setIn`: Reassigns the various "standard" streams

7

The secret life of StringBuilder

The `+` operator is overloaded to concatenate `Strings`

In reality, `javac` compiles string concatenation into operations on a `StringBuilder`:

```
double temp;
System.out.println("Today's temp is " + temp);
```

Is really:

```
double temp;
StringBuilder sb = new StringBuilder();
sb.append("Today's temp is ");
sb.append(temp);
System.out.println(sb.toString());
```

So, remember that string concatenation creates a `StringBuilder`

- There is some overhead, so don't do it inside a tight loop
- Sometimes it is better to use a `StringBuilder` directly instead of concatenation

6

java.lang.Math

`Math` provides static methods for performing mathematical calculations on `doubles`

- `abs`, `sqrt`
- Trigonometric functions (`cos`, `tan`, `asin`, et. al.)
- `ceil`, `floor`
- `exp`, `pow`
- `min`, `max`
- `toDegrees`, `toRadians`
- `random`: Returns a random `double` between 0.0 and 1.0

8

The “wrapper” classes

Some things, such as they keys of a hash table, can only be Objects.

What if you wanted to key a hash table on an `int` value?

Java provides “wrapper” classes for all of the primitive types: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, `Void`

Each wrapper class has a method that returns the value of the primitive class represents: `intValue`, `charValue`, etc.

9

`java.lang.Character`

Java supports the 16-bit Unicode standard for international character sets

A number of useful static methods

- `digit`: Returns the numeric (`int`) value of a `char`
- `forDigit`: Returns the `char` value for a number
- `isDigit`: Determines if a `char` is a digit
- `isLetter`: Determines if a `char` is a letter
- `isWhitespace`: Determines if a `char` is whitespace

Instance methods

- `charValue`: Returns the `char` value of a `Character`
- `compareTo`: Compares one `Character` to another

11

`java.lang.Boolean`

Class methods

- `valueOf`: Parses a `String` and returns its `boolean` value (case insensitive)

Instance methods

- `booleanValue`: Returns the `boolean` value for a `Boolean`
- `equals`: Compares this `Boolean` to another

10

`java.lang.Number`

The numeric wrapper classes are subclasses of `Number`

Instance methods for converting between numeric types

- `byteValue`: Returns a `Number`'s value as a `byte`
- `doubleValue`
- `floatValue`
- `intValue`
- `longValue`
- `shortValue`

All of `Number`'s subclass have similar behavior

12

Static methods:

- `parseInt`: Converts a `String` to an `int`
- `toBinaryString` Returns the binary representation of an `int` as a `String`
- `toHexString`
- `toOctalString`

Static fields:

- `MAX_VALUE`: The largest `int`
- `MIN_VALUE`: The smallest `int`

`java.lang.Throwable` is the base class for all exceptions

- `getMessage`: Returns a `String` message describing `Throwable` object
- `printStackTrace`: Prints a stack trace describing where in the code the `Throwable` was thrown
- JDK 1.4 added a `getCause` method that returns a `Throwable` that caused the other `Throwable` (chained exceptions) and a `getStackTrace` method that returns a representation of the location at which the throwable was thrown

`Throwable` has two subclasses

- `java.lang.Exceptions` are the kinds of things a reasonable program may want to catch
- `java.lang.Errors` are truly not expected (e.g. running out of virtual memory), could happen at any time, and should not be caught

Big Bucket o' java.lang Exceptions

Exceptions

- `ArithmeticException` e.g. divide by zero
- `ArrayIndexOutOfBoundsException`
- `ClassCastException` Trying to cast an object to a type that it is not
- `IllegalArgumentException`
- `NegativeArraySizeException`
- `NullPointerException` Referencing an object that is null
- `NumberFormatException` Thrown when parsing numbers

Errors

- `OutOfMemoryError` Garbage collected heap is full
- `StackOverflowError` Too much recursion

Checked versus Unchecked Exceptions

`java.lang.RuntimeExceptions` are often thrown by the Java Virtual Machine's runtime system

- Called "unchecked" exceptions because they do not need to be declared in a `throws` clause, nor do they have to be caught
- Examples include `ClassCastException` and `NullPointerException`
- Often easy to test for: "Look before you leap"

Other subclasses of `java.lang.Exception` must be explicitly thrown and caught

- These are "checked" exceptions
- Examples include `java.io.IOException`, `java.sql.SQLException`, and `java.awt.AWTException`
- Your exceptions should subclass `Exception`
- Checked exceptions make your code more explicit and easier to understand

Catching Multiple Kinds of Exceptions

A try block can have multiple catch blocks

- The type of the exception determines which catch block will be executed

```
try {
    openFile();
} catch (FileNotFoundException ex) {
    // prompt user for new file name
} catch (IOException ex) {
    // Print out error message and exit
}
```

Note that the catch statements have to be arranged according to the exception class hierarchy

```
try {
    openFile();
} catch (IOException ex) {
} catch (FileNotFoundException ex) {
    // Unreachable code. Won't compile.
}
```

17

Using Assertions vs. Throwing Exceptions

Assertions should be used to verify the internal logic of a method

An exception (such as `IllegalArgumentException`) should be used to verify the inputs to a (public) method

- Remember, it is reasonable for a program to catch an `Exception`, but it shouldn't catch an `Error`

Using assertions:

```
public void setPort(int port) {
    if (port <= 1024) {
        throw new IllegalArgumentException();
    }
}

private int readPort() {
    int port = ...; // Read from config file
    assert port > 1024 : port;
}
```

19

Assertions

JDK 1.4 added an assertion facility to the Java language

- `assert Expression1`
- `assert Expression1 : Expression2`

If `Expression1` evaluates to false, then a `java.lang.AssertionError` is thrown

- `Expression2` is the detail message issued with the `AssertionError`

Assertions are used to verify that certain program facts are true

- For instance, after reading all of the bytes from a buffer, assert that the buffer is empty

Assertions incur some runtime expense, so they must be explicitly enabled

- Assertions are enabled via the `-ea` switch to java
- Code executed by the assertion must have no side effects (e.g. changing the state of an object)

18

Assertions and Program Logic

Assertions are most useful to verify program logic

```
private String getDayString(int day) {
    switch (day) {
        case MONDAY:
            return "Monday";
        case TUESDAY:
            return "Tuesday";
        // ...
        default:
            assert false : "Unknown day: " + day;
    }
}

if (i % 3 == 0) {
    // ...
} else if (i % 3 == 1) {
    // ...
} else {
    assert i % 3 == 2;
    // ...
}
```

Using asserts will make your code better!

20

Cloning Objects

Object's `clone` method returns a copy of an object

The copy usually has the same state and commonly

```
x.clone().equals(x)
```

But obviously,

```
x.clone() != x
```

By default, the JVM doesn't know how to make a copy of an object

- By default, the `clone` method throws a `CloneNotSupportedException`

If a class implements the `Cloneable` interface, invoking the `clone` method will automatically return a *shallow copy* of the receiving object

- JVM allocates a new instance of the class of the receiver – no constructor is invoked
- Fields of the clone have the same values as the receiver object
- Contents of the fields are not cloned (clone will refer to the same objects as the original)

21

Covariant Returns

In J2SE 1.5 methods may have *covariant returns*

- An overriding method may modify the return type of a method to be a subclass of the overridden method's return type

From `edu/pdx/cs410J/j2se15/CovariantReturns.java`

```
static abstract class Animal implements Cloneable {
    public abstract Object clone();
}
```

```
static class Human extends Animal {
    public Human clone() {
        return new Human();
    }
}
```

```
static class Student extends Human {
    public Student clone() {
        return new Student();
    }
}
```

23

Cloning Objects

In order to get a *deep copy*, `clone` should be overridden:

```
public class Grades implements Cloneable {
    private double[] grades;

    public Object clone() {
        Grades grades2 = (Grades) super.clone();
        grades2.grades = this.grades.clone();
        return grades2;
    }
}
```

Some notes:

- Invoking `super.clone()` creates a new object
- Arrays are cloneable (because they are `Objects`)
- Even though the overridden clone doesn't declare that it throws `CloneNotSupportedException`, it still has to be caught
 - Superclass (`Object`) may throw it – can't change the contract

22

Covariant Returns

If you were to decompile the class files you would see

- In `Human`'s class file the declared return type of `clone` is still `Object`
 - Binary compatibility with older code
- However, a **call** to `Human.clone()` is typed as returning a `Human`

Have to be careful with using covariant returns with third-party code

- If someone else subclassed the JDK 1.4 `Human`

```
class Professor extends Human {
    public Object clone() {
        return new Professor();
    }
}
```

The code wouldn't compile because `Object` is not a subclass of `Human`

24

Covariant returns of internal classes

Very often, applications have “external” APIs and “internal” APIs

- External APIs are for users (interfaces and abstract classes)

```
package com.college;

public interface Classroom { ... }

public interface University {
    public Classroom[] getClassrooms();
}
```

- Internal APIs contain implementation (classes)

```
package com.college.internal;

public class ClassroomImpl implements Classroom {

    public class UniveristyImpl implements University
        public ClassroomImpl[] getClassrooms() { ... }
}
```

If the internal classes return internal types, the implementation code doesn't have to cast

25

The java.io package

The classes and interfaces in the java.io package provide a myriad of facilities for performing I/O operations.

- File class that represents a file
- Classes for byte-based I/O (Streams)
- Classes for text-based I/O (Readers/Writers)

26

java.io.File

File represents a file and can be created from a String specifying its location or a File representing the directory that a named file resides in.

- canWrite: Determines whether or not a File can be written to
- delete: Deletes a File
- exists: Determines if a File exists
- getName: Returns the name of a File
- isDirectory: Determine if a File represents a directory
- length: Returns the size of a File
- mkdir: Creates the directory that a File represents
- getParentFile: Returns the directory containing this File as a File

27

Other File goodies

File has four important static fields

- separator/separatorChar: The string/character that separates portions of a file spec (/ on UNIX)
- pathSeparator/pathSeparatorChar: The string/character that separates directories in a path (: on UNIX)

The java.io package contains two interfaces, FileFilter and FilenameFilter, which have an accept method that accepts/rejects a File based on some criteria (e.g. its name).

The filters are used as arguments to File's list and listFiles methods.

- list(FilenameFilter) returns the names of all files that are accepted by a FilenameFilter
- listFiles(FileFilter) returns all of the Files that are accepted by a FileFilter

28

Example using Files and filters

```
package edu.pdx.cs410J.core;
import java.io.*; // Must be imported

public class DirectoryFilter implements FileFilter {
    public boolean accept(File file) {
        if (file.isDirectory()) {
            return true;
        } else {
            return false;
        }
    }
}
```

```
package edu.pdx.cs410J.core;
import java.io.*;

public class JavaFilenameFilter
    implements FilenameFilter {
    public boolean accept(File dir, String fileName) {
        if (fileName.endsWith(".java")) {
            return true;
        } else {
            return false;
        }
    }
}
```

29

Example using Files and filters

```
package edu.pdx.cs410J.core;
import java.io.*;

public class FindJavaFiles {
    private static FileFilter    dirFilter;
    private static FilenameFilter javaFilter;

    private static void findJavaFiles(File dir) {
        File[] javaFiles = dir.listFiles(javaFilter);
        for (int i = 0; i < javaFiles.length; i++)
            System.out.println(javaFiles[i].toString());
        File[] dirs = dir.listFiles(dirFilter);
        for(int i = 0; i < dirs.length; i++)
            findJavaFiles(dirs[i]);
    }

    public static void main(String[] args) {
        File file = new File(args[0]);
        if (file.isDirectory()) {
            dirFilter = new DirectoryFilter();
            javaFilter = new JavaFilenameFilter();
            findJavaFiles(file);
        } else {
            System.err.println(file +
                " is not a directory");
        }
    }
}
```

30

Example using Files and filters

```
$ cd ~whitlock/public_html/src
$ java -cp ~/classes edu.---.FindJavaFiles .
./edu/pdx/cs410J/AbstractAirline.java
./edu/pdx/cs410J/AbstractFlight.java
./edu/pdx/cs410J/AirportNames.java
./edu/pdx/cs410J/ParserException.java
./edu/pdx/cs410J/lang/Animal.java
./edu/pdx/cs410J/lang/Ant.java
./edu/pdx/cs410J/lang/Bee.java
./edu/pdx/cs410J/lang/Bird.java
./edu/pdx/cs410J/lang/Cow.java
./edu/pdx/cs410J/lang/DivTwo.java
...
./edu/pdx/cs410J/family/TextDumper.java
./edu/pdx/cs410J/family/Parser.java
./edu/pdx/cs410J/family/TextParser.java
./edu/pdx/cs410J/family/AddPerson.java
./edu/pdx/cs410J/family/NoteMarriage.java
```

31

Why is the FileFilter interesting?

Instance of `DirectoryFilter` and `JavaFilenameFilter` do not have any state (fields)

- An object encapsulates **behavior**

The responsibility of filtering files is partitioned between the `File` API and your code:

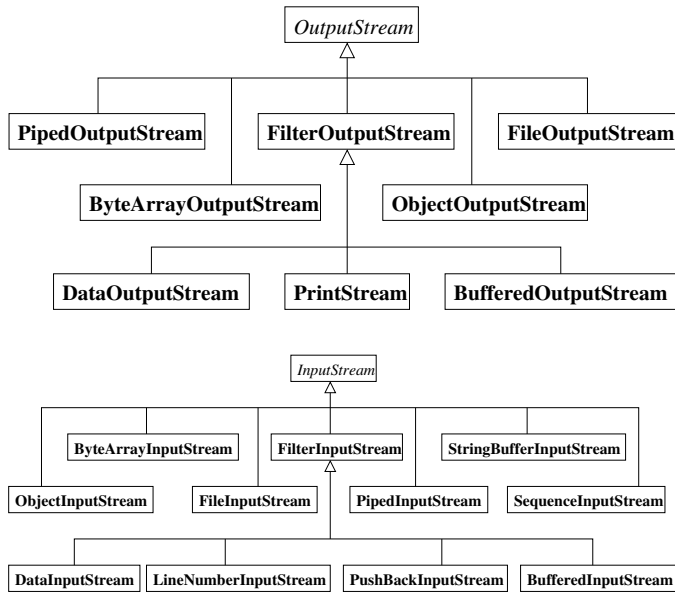
- `File` knows how to apply the filter, but doesn't know the criteria under which to filter
- You know what you want to filter, but `File` takes care of doing the grunt work

`File` **delegates** some of its work to the filter's `accept` method

32

Streams: I/O in bytes

The `java.io` package in JDK 1.0 contained two hierarchies of classes for performing byte-based stream I/O



33

`java.io.OutputStream`

An `OutputStream` is an abstract class that writes bytes and has the following methods:

- `write`: Writes bytes to the stream
- `close`: Closes the stream and releases any resources associated with it
- `flush`: Sends all pending output to the stream

Some `OutputStreams`

- `ByteArrayOutputStream`: Writes to a byte array
- `PipedOutputStream`: Used with a `PipedInputStream` to send data between threads
- `ObjectOutputStream`: Writes Objects to a stream

34

`java.io.FileOutputStream`

A `FileOutputStream` write bytes to a file

Constructed from a `File` or a file's name, may throw a `FileNotFoundException`

`java.io.FilterOutputStream`

A `FilterOutputStream` is built around another `OutputStream` and performs some processing on its bytes

- `BufferedOutputStream`: Buffers the data to be written
- `DataOutputStream`: Writes Java's primitive types in a machine-independent format
- `PrintStream`: Writes data in a human-readable format, doesn't throw exceptions
 - `System.out` and `System.err` are `PrintStreams`
 - Has `print` and `println` methods for all types
 - The `hasError` method determines if an error has occurred

35

An example using `OutputStreams`

```
package edu.pdx.cs410J.core;
import java.io.*;

public class WriteDoubles {
    static PrintStream err = System.err;

    public static void main(String[] args) {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(args[0]);
        } catch (FileNotFoundException ex) {
            err.println("** No such file: " + args[0]);
            System.exit(1);
        }
        DataOutputStream dos = new DataOutputStream(fos);
        for(int i = 1; i < args.length; i++) {
            try {
                double d = Double.parseDouble(args[i]);
                dos.writeDouble(d);
            } catch (NumberFormatException ex) {
                err.println("** Not a double: " + args[i]);
            } catch (IOException ex) {
                err.println("** " + ex);
                System.exit(1);
            }
        }
    }
}
```

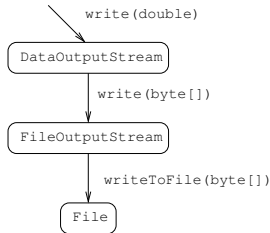
36

An example using OutputStreams

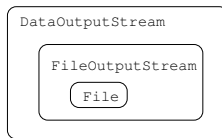
```
$ java -cp ~/classes edu.---.WriteDoubles \  
  doubles.out 1.23 2.34 3.45
```

If you were to `cat doubles.out` you would see garbage because `doubles.out` is a binary file.

Behavior Delegation



Object Composition (the "object onion")



37

java.io.FileInputStream

`FileInputStream` is used for reading bytes from a file

Constructed from a `File` or a file's name, may throw a `FileNotFoundException`

java.io.FilterInputStream

A `FilterInputStream` is built around another `InputStream` and processes the bytes that are read

- `BufferedInputStream`: Buffer the input read from another `InputStream`
- `DataInputStream`: Used to read Java's primitive types
- `PushbackInputStream`: Allows you to push bytes back into the stream

39

java.io.InputStream

`InputStreams` read bytes and have the following methods:

- `available`: Returns the number of bytes that can be read without blocking
- `close`: Closes the stream
- `read`: Reads bytes into a byte array. Returns the number of bytes read, -1 if done.
- `skip`: Skips over some number of bytes in the stream

Some `InputStreams`:

- `ByteArrayInputStream`: `InputStream` behavior over a byte array
- `PipedInputStream`: Used with a `PipedOutputStream` to send data between threads
- `SequenceInputStream`: Read from multiple `InputStreams` in a given order

38

An example using InputStreams

```
package edu.pdx.cs410J.core;  
import java.io.*;  
public class ReadDoubles {  
    static PrintStream out = System.out;  
    static PrintStream err = System.err;  
    public static void main(String args[]) {  
        FileInputStream fis = null;  
        try {  
            fis = new FileInputStream(args[0]);  
        } catch (FileNotFoundException ex) {  
            err.println("** No such file: " + args[0]);  
        }  
        DataInputStream dis = new DataInputStream(fis);  
        while (true) {  
            try {  
                double d = dis.readDouble();  
                out.print(d + " ");  
                out.flush();  
            } catch (EOFException ex) {  
                break; // All done reading  
            } catch (IOException ex) {  
                err.println("** " + ex);  
                System.exit(1);  
            }  
        }  
        out.println("");  
    }  
}
```

40

An example using `InputStreams`

```
$ java -cp ~/classes edu.---.ReadDoubles \  
  doubles.out  
1.23 2.34 3.45
```

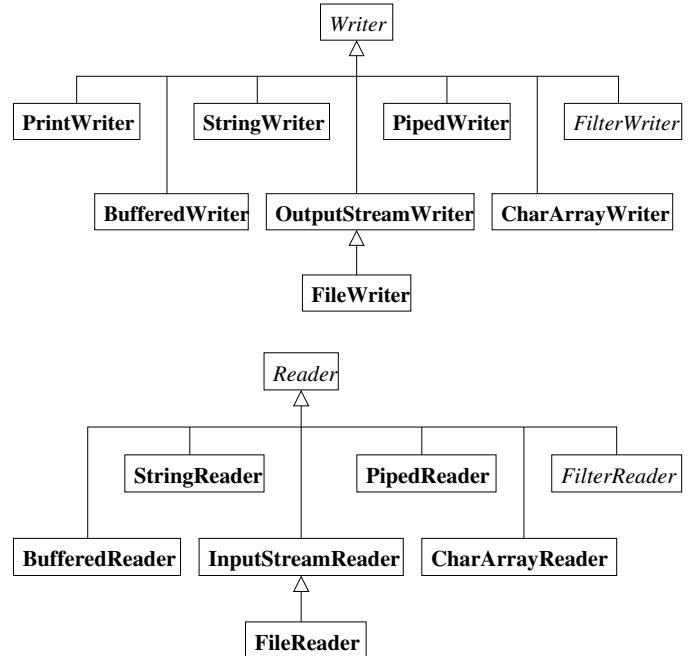
There's no nice way of telling when a `DataInputStream` is done – have to catch an `EOFException` – yuch!

Note the use of `print` and `flush`

41

Handling text I/O: Writers and Readers

Streams worked well for byte data, but working with text data was awkward. JDK 1.1 introduced writers and readers:



42

`java.io.Writer`

`Writer` is an abstract class and writes characters to some destination. It has methods such as

- `write`: Writes characters or strings
- `close`: Closes a `Writer`
- `flush`: Sends all pending text to the destination

Some `Writers`

- `BufferedWriter`: Buffers text before writing it to the destination
- `CharArrayWriter`: Writes text to a char array
- `FilterWriter`: Abstract class for writing filtered text streams
- `PipedWriter`: Used with a `PipedReader` to send text between threads
- `OutputStreamWriter`: Converts chars to bytes and sends them to an `OutputStream`

43

`java.io.PrintWriter`

A `PrintWriter` prints formatted text to another `Writer` or an `OutputStream`

Like a `PrintStream` in that it has `print` and `println` methods, but flushing is not automatic

`java.io.StringWriter`

`StringWriter` is a `Writer` that writes to a `String`

- `getBuffer`: Returns the `StringBuilder` written to
- `toString`: Returns the `String` being written to

`java.io.FileWriter`

A `FileWriter` writes text to a file

The file is specified by a `File` object or the file's name

44

Example using Writers

```
package edu.pdx.cs410J.core;
import java.io.*;

public class WriteToFile {
    private static PrintWriter err;

    public static void main(String[] args) {
        // Wrap a PrintWriter around System.err
        err = new PrintWriter(System.err, true);

        try {
            Writer writer = new FileWriter(args[0]);

            // Write command line arguments to the file
            for(int i = 1; i < args.length; i++) {
                writer.write(args[i]);
                writer.write('\n');
            }

            // All done
            writer.flush();
            writer.close();

        } catch(IOException ex) {
            err.println("** " + ex);
        }
    }
}
```

45

Example using Writers

```
$ java -cp ~/classes edu.---.WriteToFile \
    text.out This is some text
$ cat text.out
This
is
some
text
```

Note how we “wrapped” a `PrintWriter` around a `PrintStream`

This abstraction helps simplify programming by hiding what’s really going on

You don’t know what you’re writing to and, more importantly, you don’t care!

46

`java.io.Reader`

`Reader` is an abstract class for reading character data from a source

- `read`: Reads chars
- `ready`: Determines if a `Reader` has more text to read
- `close`: Closes a `Reader`
- `skip`: Skips some number of characters

Some `Reader`s

- `CharArrayReader`: Reads from a char array
- `FilteredReader`: Abstract class for reading filtered character streams
- `PipedReader`: Used with a `PipedWriter` to send text between threads
- `StringReader`: Reads from a `String`
- `InputStreamReader`: Reads from an `InputStream`
- `BufferedReader`: Buffers the text it reads
 - Has a `readLine` method

47

Example using Readers

```
package edu.pdx.cs410J.core;
import java.io.*;

public class ReadFromConsole {
    public static void main(String[] args) {
        InputStreamReader isr =
            new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        StringWriter text = new StringWriter();

        while (true) {
            try {
                // Read a line from the console
                String line = br.readLine();

                if (line.equals("-1")) {
                    break;
                } else {
                    text.write(line + " ");
                }
            } catch(IOException ex) {
                System.err.println("** " + ex);
                System.exit(1);
            }
        }
        System.out.println(text);
    }
}
```

48

Example using Readers

```
$ java -cp ~/classes edu.---.ReadFromConsole
Does
this
program
work?
-1
Does this program work?
```

49

Closing Streams

To free up system resources, streams (and readers/writers) should be closed by invoking their `close` method, often in a `finally` block

- It's easy to forget to call `close`
- And `close` may throw an `IOException`

```
public void printTextFile(File file)
    throws IOException {

    BufferedReader br =
        new BufferedReader(new FileReader(file));
    try {
        while (br.ready()) {
            System.out.println(br.readLine());
        }

    } catch (IOException ex) {
        System.err.println(ex);
        throw ex;
    } finally {
        if (br != null) {
            br.close();
        }
    }
}
```

50

Automatically Closing Streams

In Java 7, a new “try with resources” language feature was added: A `try` statement can declare instances of `java.lang.AutoCloseable` that are automatically closed when the `try` block is exited

- The `java.io.Closeable` interface that is implemented by most I/O classes subclasses `AutoCloseable`

```
public void printTextFile(File file)
    throws IOException {

    try (BufferedReader br =
        new BufferedReader(new FileReader(file))) {
        while (br.ready()) {
            System.out.println(br.readLine());
        }

    } catch (IOException ex) {
        System.err.println(ex);
    }
}
```

Now you don't need to remember to `close` the reader

51

The utility classes

The `java.util` package contains a number of useful and handy classes

- `StringTokenizer`, `Vector`, `Hashtable`, `Stack`
- The collection classes
- `Date`, `Calendar`, `Locale`
- System properties

52

java.util.StringTokenizer

A StringTokenizer is used to parse a String*

The constructor takes the String to be parsed and a String whose characters delimit tokens (by default whitespace delimits tokens)

- `countTokens`: Returns the number of tokens remaining
- `nextToken`: Returns the next token in the String
- `hasMoreTokens`: Are there more tokens to be returned?

*JDK 1.4 added a regular expression package to Java (`java.util.regex`) that provides Perl-like regex

53

StringTokenizer example

```
package edu.pdx.cs410J.core;

import java.util.*;

public class ParseString {
    /**
     * The second <code>String</code> from the
     * command line contains the parsing delimiters.
     */
    public static void main(String[] args) {
        String string = args[0];
        String delimit = args[1];
        StringTokenizer st;
        st = new StringTokenizer(string, delimit);

        while (st.hasMoreTokens()) {
            String token = st.nextToken();
            System.out.println("Token: " + token);
        }
    }
}

$ java -cp ~/classes edu.---.ParseString \
  This,is,a:sentence. ,:
Token: This
Token: is
Token: a
Token: sentence.
```

54

The Original Collection Classes

The first Java release contained several classes for collecting objects together:

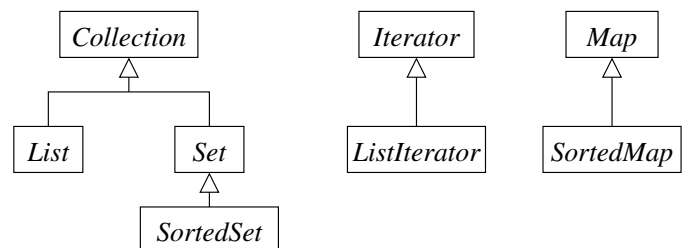
- `Vector`: A growable, ordered collection of objects
- `Stack`: A `Vector` with push/pop
- `Hashtable`: Maps objects to objects

While these classes were very useful, they tended to be bulky and slow.

55

Collection Classes

First of all, a hierarchy of interfaces in `java.util`



`java.util.Collection` groups objects together

- `add`: Adds an Object to a Collection
- `contains`: Determines if a Collection contains an Object
- `isEmpty`: Determines if a Collection is empty
- `iterator`: Returns an Iterator over a Collection
- `remove`: Removes an Object from a Collection
- `size`: Returns the number of elements in a Collection

56

java.util.List

The elements of a `List` are 0-indexed

- `add`: Adds an `Object` at a given index
- `get`: Returns the `Object` at a given index
- `set`: Sets the `Object` at a given index
- `listIterator`: Returns a `ListIterator` over a `List`

java.util.Set

Sets are unordered and each element in a `Set` is unique

The `equals` method is used to determine the equality of two `Objects`

java.util.SortedSet

A `Set` whose elements are ordered

Has methods like `first` and `last`

57

java.util.Iterator

An `Iterator` is used to iterate over the `Objects` in a collection

- `hasNext`: Determines if there are any more elements to be iterated over
- `next`: Returns the next element to be examined
- `remove`: Removes the element returned by `next` from the underlying collection (not always implemented)

java.util.ListIterator

`ListIterators` can iterate in both directions

- `add`: Inserts an `Object` into the underlying list
- `hasPrevious`: Determines whether or not there is a previous element in the list
- `previous`: Returns the previous element in the underlying list
- `nextIndex/previousIndex`: Returns the index of the element that would be returned by `next/previous`

58

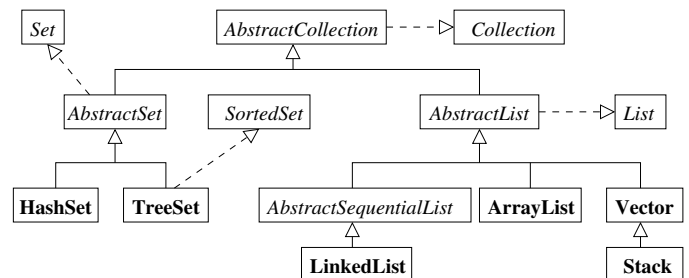
java.util.Map

A `Map` maps *key* objects to *value* objects

- `put`: Creates a mapping from one `Object` to another in a `Map`
 - Invokes the key's `hashCode` method
- `get`: Returns the `value Object` associated with a given `key Object`
- `containsKey`: Determines if an `Object` is a key in the mapping
- `containsValue`: Determines if an `Object` is a value in the mapping
- `keySet`: Returns the keys in a `Map` as a `Set`
- `values`: Returns the values in a `Map` as a `Collection`
- `entrySet`: Returns the mappings in a `Map` as a `Set`

59

Abstract collection classes



To ease the implementation of collection classes, several abstract base classes are provided:

- `java.util.AbstractCollection`
- `java.util.AbstractList`: Backed by a random access data structure (e.g. array)
- `java.util.AbstractSequentialList`: Backed by a sequential access data structure (e.g. linked list)
- `java.util.AbstractMap`
- `java.util.AbstractSet`

60

Concrete implementations of collections

In the `java.util` package: Lists

- `ArrayList`: List backed by an array
- `LinkedList`: List back by a linked list, provides stack-like behavior
- `Vector`: Implements the `List` interface

Maps

- `HashMap`: Constant-time `get` and `put`
- `TreeMap`: Sorted keys gives $\log(n)$ `get` and `put`
- `IdentityHashMap`: Key comparison based on identity (`==`) instead of `equals` method
- `LinkedHashMap`: Keeps track of insertion order of mappings

Sets

- `HashSet`: Set backed by a hash table
- `TreeSet`: `SortedSet` backed by a red-black tree

61

Working with our example

```
$ java -cp ~/classes edu.---.Collections
One
Two
Three

One
Three
Four
Two
```

Note order of `ArrayList` and that a `HashSet` contains unique values

Abstraction is key: "Program to the interface"

63

Example using collections

```
package edu.pdx.cs410J.core;
import java.util.*; // Must be imported!

public class Collections {

    /** Prints the contents of a Collection */
    private static void print(Collection c) {
        Iterator iter = c.iterator();
        while (iter.hasNext()) {
            Object o = iter.next();
            System.out.println(o);
        }
    }

    public static void main(String[] args) {
        Collection c = new ArrayList();
        c.add("One");
        c.add("Two");
        c.add("Three");
        print(c);
        System.out.println("");

        Set set = new HashSet(c);
        set.add("Four");
        set.add("Two");
        print(set);
    }
}
```

62

Storing primitives in collection

Collections take `Objects`, but `ints`, `doubles`, `booleans`, etc. are not `Objects`

Use the wrapper classes to create `Objects` that represent the primitives:

```
package edu.pdx.cs410J.core;
import java.util.*;

public class WrapperObjects {
    public static void main(String[] args) {
        Collection c = new ArrayList();
        c.add(new Integer(4));
        c.add(new Double(5.3));
        c.add(new Boolean(false));

        System.out.println(c);
    }
}
```

64

Autoboxing of primitive types

J2SE 1.5 provides automatic conversion of primitives to wrapper objects in a procedure called “autoboxing”

- Autoboxing is applied to variable and field assignments, the arguments to method calls, and casting

```
package edu.pdx.cs410J.j2se15;
import java.util.*;

public class Autoboxing {
    public static void main(String[] args) {
        // Note that Integer.valueOf returns an Integer
        int i = Integer.valueOf("123");

        List list = new ArrayList();
        list.add(i);
        int j = (Integer) list.get(0);
    }
}
```

65

Generics add some complexity to the type system

Even though a `String` is an `Object`, an `ArrayList<String>` is **not** a `List<Object>`

Otherwise, you could do this:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Integer(42)); // Bad!
```

Because the compiler cannot determine that `lo` may actually only contain `Strings`, the language disallows the assignment.

67

Strongly typing collections

Originally, collections could only contain `Objects`

“Generic types” introduced in Java 5 allow you to specify the type of objects that a collection may contain

- `List<String>` is pronounced “a list of strings”
- `List<Long> longs = new ArrayList<Long>();`
- Attempting to a non-`Long` to `longs` will caused a compilation error:
 - `longs.add("This will not compile")`

66

Generics added some complexity to the language

Being forced to include all of the generic types in a variable declaration made for hard-to-read code:

```
List<Map<String, String>> data =
    new ArrayList<Map<String, String>>();
```

Java 7 introduced the generics “diamond” that infers the generic types on the left side of the assignment:

```
List<Map<String, String>> data = new ArrayList<>();
```

68

Iterating over collections

What happens when a collection is modified while it is being iterated over?

```
package edu.pdx.cs410J.core;
import java.util.*;

public class ModifyWhileIterating {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("one"); list.add("two");

        Iterator<String> iter = list.iterator();
        while (iter.hasNext()) {
            String s = iter.next();
            if (s.equals("one")) {
                list.add(0, "start");
            }
        }
    }
}

$ java -cp ~/classes edu.---.ModifyWhileIterating
Exception in thread "main"
    java.util.ConcurrentModificationException
```

69

Iterators and the enhanced for loop

The enhanced for loop syntax can be used with `Collections*` as well as arrays

```
Collection coll = ...
for (Object o : coll) {
    System.out.println(o);
}
```

See `edu.pdx.cs410J.j2se15.EnhancedForLoop`

This syntax is compact, but you cannot reference the Iterator object

- Can't remove an element from the Collection while you're iterating over it

*Actually, any object that implements the `java.lang.Iterable` interface

71

Iterating over collections

Most Iterators are *fail-fast*

- If the underlying collection is modified (e.g. it size changes), then subsequent calls to `next` will result in a `ConcurrentModificationException`
- To safely modify an underlying collection, use Iterator's `remove` method

Fail-fast iterators have the benefit of immediately detecting when they are out-of-date

- Iterator fails quickly instead of allowing potential non-deterministic (or simply incorrect) behavior

However, you should not rely on a `ConcurrentModificationException` always being thrown:

- Replacing an item in a `List` (using `put`) may not cause the iterator to fail
- Fail-fast behavior should only be used to detect bugs

70

Example working with Maps

```
package edu.pdx.cs410J.core;

import edu.pdx.cs410J.lang.*;
import java.util.*;

public class Farm {
    /** Prints the contents of a Map. */
    private static void print(Map<String, Animal> map)
        for (String key : map.keySet()) {
            Animal value = map.get(key);
            String s = key + " -> " + value;
            System.out.println(s);
        }
}

public static void main(String[] args) {
    Map<String, Animal> farm = new HashMap<>();
    farm.put("Old MacDonald",
            new Human("Old MacDonald"));
    farm.put("Bossie", new Cow("Bossie"));
    farm.put("Clyde", new Sheep("Clyde"));
    farm.put("Louise", new Duck("Louise"));

    print(farm);
}
```

72

Working with our Map example

```
$ java -cp ~/classes edu.---.Farm
Clyde -> Clyde says Baa
Bossie -> Bossie says Moo
Old MacDonald -> Old MacDonald says Hello
Louise -> Louise says Quack
```

Note that the order in which the elements were added to the `HashMap` has nothing to do with the order in which the `Iterator` visits them

Note also:

- Maps use the key object's `hashCode` method to determine the bucket in which to search
- Each element in the bucket's collision chain is compared to the key object using its `equals` method

So, if instances of your own classes are to be used as keys in a Map

- You should override `equals` and `hashCode`
- Note that two objects that are equal must have the same hash code

73

An example of Natural Ordering

Instances of `Cereal` are naturally sorted alphabetically by their name

```
package edu.pdx.cs410J.core;
import java.util.*;

public class Cereal implements Comparable<Cereal> {
    private String name;
    private double price;

    // <snip>

    public int compareTo(Cereal c2) {
        return this.getName().compareTo(c2.getName());
    }

    public boolean equals(Object o) {
        if (o instanceof Cereal) {
            Cereal other = (Cereal) o;
            return this.getName().equals(other.getName());
        }
        return false;
    }

    public int hashCode() {
        return this.getName().hashCode();
    }
}
```

75

Comparing Objects

Objects that implement the `java.lang.Comparable` interface are said to have a *natural ordering*

- Instances of `String`, `Integer`, `Double`, etc. are all `Comparable`
- `Comparable`'s `compareTo` method compares the receiver (x) object to another object (y)
 - if $x < y$, a negative `int` should be returned
 - if $x == y$, zero should be returned*
 - if $x > y$, a positive `int` should be returned
- `Comparable` has a generic type that specifies the class of object it can compare itself to
 - Often you compare an object to another object of its same type

Unless instructed otherwise, classes and methods that sort objects (such as `SortedSets`) will respect their natural ordering

*Should have the same semantics as the `equals` method

74

An example of Natural Ordering

```
public static void main(String[] args) {
    SortedSet<Cereal> set = new TreeSet<Cereal>();
    set.add(new Cereal("Total", 3.56));
    set.add(new Cereal("Raisin Bran", 2.65));
    set.add(new Cereal("Sugar Crisps", 2.38));

    for (Cereal c : set) {
        System.out.println(c);
    }
}
```

Running the example...

```
$ java -cp ~/PSU/src/classes edu.---.Cereal
Raisin Bran $2.65
Sugar Crisps $2.38
Total $3.56
```

Natural ordering allows the author of the class to specify how instances of that class are compared

76

Custom Sorted Collections

The `java.util.Comparator` interface is used to sort objects by criteria other than their natural ordering

- A `Comparator` specifies a *total ordering* over a set of objects
- A `Comparator`'s `compare` method compares two objects and returns an `int` with the same meaning as `Comparable`'s `compareTo` method
- A `Comparator` may or may not choose to respect the `equals` method of the objects that it is comparing
- `Comparator` has a generic type that specifies the type of object that is compared

`Comparators` can be used to create `TreeSets` and `TreeMaps`

77

An example `Comparator`

Compares boxes of `Cereal` based on their price

```
package edu.pdx.cs410J.core;

import java.util.*;

public class CerealComparator
    implements Comparator<Cereal> {
    public int compare(Cereal o1, Cereal o2) {
        double price1 = o1.getPrice();
        double price2 = o2.getPrice();

        if (price1 > price2) {
            return 1;
        } else if (price1 < price2) {
            return -1;
        } else {
            return 0;
        }
    }
}

// Continued..
```

78

An example `Comparator`

```
public static void main(String[] args) {
    Set<Cereal> set =
        new TreeSet<Cereal>(new CerealComparator());
    set.add(new Cereal("Cap'n Crunch", 2.59));
    set.add(new Cereal("Trix", 3.29));
    set.add(new Cereal("Count Chocula", 2.59));
    set.add(new Cereal("Froot Loops", 2.45));

    // Print out the cereals
    for (Cereal c : set) {
        System.out.println(c);
    }
}
```

```
$ java -cp ~/classes edu.---.CerealComparator
Froot Loops $2.45
Cap'n Crunch $2.59
Trix $3.29
```

Why wasn't `Count Chocula` printed out?

79

Helpful collection functions

The `java.util.Collections` class* contains helpful static methods for working with collections

- `max(Collection)` returns the largest element in a collection (uses natural ordering)
- `nCopies(int, Object)` returns a `List` contains `n` copies of a given object
- `singleton(Object)` returns an immutable `Set` that contains only the given object
- `sort(List, Comparator)` sorts a list using the given comparator
- `unmodifiableMap(Map)` returns a `Map` that cannot be modified that has the same contents as the input `Map`
 - Attempts to modify the `Map` throw an `UnsupportedOperationException`

*This class cannot be instantiated.

80

Helpful collection functions

The `java.util.Arrays` class contains static methods for working with arrays

- `asList(Object[])` returns a `List` that is backed by a given array
 - Changes to the list will “write through” to the backing array
- `binarySearch(int[], int)` returns the array index at which the given `int` occurs
- `equals(int[], int[])` returns whether or not two arrays have the same contents
- `fill(int[], int)` populates each element of an array with the given value
- `sort(int[])` sorts an array in-place

Each of these methods is overloaded to operate on the different kinds of arrays (`double[]`, `Object[]`, etc.)

81

An example of a type-safe enumeration

```
package edu.pdx.cs410J.j2se15;
import java.util.*;

public class EnumeratedTypes {
    private enum Day { SUNDAY, MONDAY, TUESDAY,
        WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }

    private static String enEspañol(Day day) {
        switch (day) {
            case SUNDAY:
                return "Domingo";
            case MONDAY:
                return "Lunes";
            case TUESDAY:
                return "Martes";
            case WEDNESDAY:
                return "Miercoles";
            case THURSDAY:
                return "Jueves";
            case FRIDAY:
                return "Viernes";
            case SATURDAY:
                return "Sabado";
            default:
                String s = "Unknown day: " + day;
                throw new IllegalArgumentException(s);
        }
    }
}
```

83

Type-safe enumerations

J2SE 1.5 provides an `enum` facility* that is like a class, but has a set of pre-defined instances (“constants”)

- The `enum` is similar to a class in that it has its own namespace (can be referenced via an `import static`)
- Unlike `static final` fields, the values of references are not compiled into the class
 - Can change `enum` values without having to recompile all of your code
- Have useful `toString`, `equals`, and `hashCode` methods (can be used with `Collections`)
- `enums` can implement interfaces, are `Serializable` and `Comparable`, and can be used in a `switch` statement
- Compile-time type safety (constants are no longer just ints)

*Based on Item 21 from Joshua Bloch's *Effective Java* book

82

Type-safe enumerations

```
public static void main(String[] args) {
    SortedSet<Day> set = new TreeSet<Day>();
    set.add(Day.WEDNESDAY);
    set.add(Day.MONDAY);
    set.add(Day.FRIDAY);

    System.out.print("Sorted days: ");
    for (Day day : set) {
        System.out.print(day + " ");
    }

    System.out.print("\nEn español: ");
    for (Day day : set) {
        System.out.print(enEspañol(day) + " ");
    }
    System.out.println("");
}

$ java -cp ~/classes edu.---.EnumeratedTypes
Sorted days: MONDAY WEDNESDAY FRIDAY
En español: Lunes Miercoles Viernes
```

84

Type-safe enumerations implementation

enums are compiled into Java inner classes

- All enums extend the `java.lang.Enum` class that provides methods like `equals`, `hashCode`, and `ordinal`
- The only non-final method of Enum is `toString` – the rest is taken care of for you

The compiler adds two interesting static methods to the enum class:

- `values` returns an array of each enumeration instance
- `valueOf` return the enumerated instance with the given name

```
Coin dime = Coin.valueOf("DIME");
```

85

Type-safe enumeration with added behavior

```
public static void main(String[] args) {
    Operation[] ops = { Operation.PLUS,
                       Operation.MINUS, Operation.TIMES,
                       Operation.DIVIDE };
    for (Operation op : ops) {
        System.out.println("5 " + op.getSymbol() +
                           " 2 = " + op.eval(5, 2));
    }
}
```

```
$ java -cp ~/classes edu.---.NumericOperators
5 + 2 = 7.0
5 - 2 = 3.0
5 * 2 = 10.0
5 / 2 = 2.5
```

87

Type-safe enumeration with added behavior

You can also attach additional behavior to enumerated types:

```
package edu.pdx.cs410J.j2se15;

public class NumericOperators {
    private abstract enum Operation {
        PLUS {
            double eval(double x, double y) {
                return x + y;
            }
            char getSymbol() { return '+'; }
        };

        MINUS {
            double eval(double x, double y) {
                return x - y;
            }
            char getSymbol() { return '-'; }
        };

        // Method declarations follow enumerations
        abstract double eval(double x, double y);
        abstract char getSymbol();
    }
}
```

86

java.util.Properties

Properties instances map Strings to Strings and are usually used to store configuration information about the JVM or an application.

- `setProperty`: Set a named property to a given value
- `getProperty`: Returns the value of a property with a given name
- `list`: Prints the contents of the Properties to a `PrintStream`
- `load`: Loads properties from some source (e.g. a file)
- `store`: Stores properties in a format suitable for use with `load`

Properties implements the Map interface

- Note that `put` will not complain if you add a non-String property

88

The JVM system properties

The JVM maintains a `Properties` object that contains various JVM settings known as *system properties*

System properties may be set with the `-D` option to `java`

Accessing the JVM's system properties:

- `System.getProperties`: Returns the system's `Properties` instance
- `System.getProperty`: Returns the value of a given named system property

Wrapper classes have static "get" methods that decode system properties as a given primitive type

- `Integer.getInteger`, `Boolean.getBoolean`

89

Example using system properties

```
package edu.pdx.cs410J.core;

import java.util.*;

public class SystemProperties {
    /**
     * Print out the system properties and check
     * to see if the "edu.pdx.cs410J.Debug"
     * property has been set on the command line.
     */
    public static void main(String[] args) {
        // Print out some properties
        Properties props = System.getProperties();
        props.list(System.out);

        // Is the "edu.pdx.cs410J.Debug" property set?
        String name = "edu.pdx.cs410J.Debug";
        boolean debug = Boolean.getBoolean(name);
        System.out.print("\nAre we debugging? ");
        System.out.println((debug ? "Yes." : "No."));
    }
}
```

90

Example using system properties

```
$ java -Dedu.pdx.cs410J.Debug=true -cp ~/classes \
  edu.pdx.cs410J.core.SystemProperties
-- listing properties --
java.vm.version=1.5.0-b64
java.vm.vendor=Sun Microsystems Inc.
path.separator=:
java.vm.name=Java HotSpot(TM) Client VM
user.dir=/u/whitlock/public_html/src
java.runtime.version=1.5.0-b64
os.arch=sparc
java.io.tmpdir=/var/tmp/
line.separator=

os.name=SunOS
java.class.version=49.0
os.version=5.9
user.home=/u/whitlock
edu.pdx.cs410J.Debug=true                <--
java.specification.version=1.5
user.name=whitlock
java.class.path=/u/whitlock/jars/examples.jar
java.home=/pkgs/jdk1.5/jre
user.language=en
file.separator=
```

Are we debugging? Yes.

91

java.util.Date

The `Date` class represents a date and a time as the number of milliseconds since midnight on January 1, 1970.

- `after` Determines if a `Date` occurs after another
- `before`
- `getTime` Returns the aforementioned number of milliseconds

A `Date` instantiated with the zero-argument constructor represents the current date/time.

Support for internationalization and multiple day/time formats complicates Java's day/time facility.

- `java.util.Calendar`
- `java.text.DateFormat`

See `edu.pdx.cs410J.core.AroundTheWorld`

92

A Calendar is used to get information (e.g. the day of the week) about a Date.

Calendar has a number of static int fields

- Info about days: DAY_OF_MONTH, DAY_OF_YEAR, YEAR
- Info about time: HOUR, MINUTE, SECOND

Calendar instance methods:

- setTime: Sets the Date for a Calendar
- add: Adds to one of a date's fields (e.g. MONTH)
- get: Returns the value of a date's field

All of Calendar's constructors are protected. How do we get a Calendar to work with?

Calendar's static getInstance method returns a Calendar instance.

93

Working with our Date and Calendar example

```
$ java -cp ~/classes edu.---.Today
Today is Thu Jul 28 15:31:11 PDT 2005
It's been 1122589871595ms since the epoch.
It is the 5th day of the week
and the 209th day of the year.
We are in the 5th week of the month.
```

The fact that the representation of a date (Date) is separate from how it is accessed (via a Calendar) makes Java's time facility more modular.

Different Calendars can treat time differently

- Gregorian calendar
- Hebrew calendar
- Chinese calendar

95

An example using Date and Calendar

```
package edu.pdx.cs410J.core;

import java.util.*;

public class Today {
    public static void main(String[] args) {
        Date today = new Date();
        Calendar cal = Calendar.getInstance();
        cal.setTime(today);

        int dayOfWeek = cal.get(Calendar.DAY_OF_WEEK);
        int dayOfYear = cal.get(Calendar.DAY_OF_YEAR);
        int weekOfMonth =
            cal.get(Calendar.WEEK_OF_MONTH);

        StringBuilder sb = new StringBuilder();
        sb.append("Today is " + today + "\n");
        sb.append("It's been " + today.getTime() +
            "ms since the epoch.");
        sb.append("\nIt is the " + dayOfWeek +
            "th day of the week \nand the " +
            dayOfYear + "th day of the year. ");
        sb.append("\nWe are in the " + weekOfMonth +
            "th week of the month.");

        System.out.println(sb.toString());
    }
}
```

94

java.text.DateFormat

The DateFormat class is used to format Dates into Strings (format) and convert Strings into Dates (parse).

- DateFormat.SHORT: 6/17/94 9:37 PM
- DateFormat.MEDIUM: Jun 17, 1994 9:37:45 PM
- DateFormat.LONG: June 17, 1994 9:37:45 PM PDT
- DateFormat.FULL: Friday, June 17, 1994 9:37:45 PM PDT

Like Calendar, you use static methods to get an instance of DateFormat

- getTimeInstance: Returns a DateFormat for formatting/parsing a time (9:37 PM)
- getDateInstance: Returns a DateFormat for formatting/parsing a date (6/17/94)
- getDateTimeInstance: Returns a DateFormat for formatting/parsing both a date and time (6/17/94 9:37 PM)
- setLenient: Sets how strict parsing should be

96

Working with DateFormat

```
package edu.pdx.cs410J.core;

import java.text.*;
import java.util.*;

public class FormattedDate {
    public static void main(String[] args) {
        // Glue args together into one String
        StringBuilder sb = new StringBuilder();
        for(int i = 0; i < args.length; i++) {
            sb.append(args[i] + " ");
        }

        Date date = null;
        int f = DateFormat.MEDIUM;

        DateFormat df =
            DateFormat.getDateInstance(f, f);

        try {
            date = df.parse(sb.toString().trim());
        } catch(ParseException ex) {
            System.err.println("** Bad date: " + sb);
            System.exit(1);
        }

        // Continued...
```

97

Working with DateFormat

```
f = DateFormat.SHORT;
df = DateFormat.getDateInstance(f, f);
System.out.println("SHORT: " + df.format(date));

f = DateFormat.MEDIUM;
df = DateFormat.getDateInstance(f, f);
System.out.println("MEDIUM: " + df.format(date));

f = DateFormat.LONG;
df = DateFormat.getDateInstance(f, f);
System.out.println("LONG: " + df.format(date));

f = DateFormat.FULL;
df = DateFormat.getDateInstance(f, f);
System.out.println("FULL: " + df.format(date));
}
}

$ java -cp ~/classes edu.---.FormattedDate \
    Jun 17, 1994 9:37:45 PM
SHORT: 6/17/94 9:37 PM
MEDIUM: Jun 17, 1994 9:37:45 PM
LONG: June 17, 1994 9:37:45 PM PDT
FULL: Friday, June 17, 1994 9:37:45 PM PDT
```

98

A more flexible format: SimpleDateFormat

`java.text.SimpleDateFormat` lets you specify a String that specifies the format of the date to parse/format

Symbol	Meaning	Presentation
G	era	Text
y	year	Number
M	month in year	Text & Number
d	day in month	Number
h	hour in am/pm (1-12)	Number
H	hour in day (0-23)	Number
m	minute in hour	Number
s	second in minute	Number
S	millisecond	Number
E	day in week	Text
D	day in year	Number
F	day of week in month	Number
w	week in year	Number
W	week in month	Number
a	am/pm marker	Text
k	hour in day (1-24)	Number
K	hour in am/pm (0-11)	Number
z	time zone	Text
'	escape for text	Delimiter
''	single quote	Literal

99

Using SimpleDateFormat

```
package edu.pdx.cs410J.core;

import java.text.*;
import java.util.*;

public class SimpleDate {
    public static void main(String[] args) {
        DateFormat df = new SimpleDateFormat(args[0]);
        Date now = new Date();
        System.out.println(df.format(now));
    }
}
```

Alphabetical characters must be escaped:

```
$ java edu.---.SimpleDate "E M d, y G 'at' h:mm a z"
Sun 4 29, 01 AD at 3:59 PM PDT
```

The more times a symbol occurs in the format string, the more verbose the format:

```
$ java edu.---.SimpleDate \
    "EEEE MMM d, yyyy G 'at' h:mm a zzzz"
Sunday Apr 29, 2001 AD at 3:59 PM Pacific Daylight Ti
```

100

Many kinds of DateFormats

Again we've seen how the presentation of a date (`DateFormat`) is separated from the date itself (`Date`).

This mechanism allows us to display dates in a variety of ways.

The `java.util.Locale` class represents a certain language/country combination.

There is a `DateFormat` for each `Locale` that parses and formats dates according to the local convention.

For instance, in the `FRANCE` locale, a date is printed as:

```
samedi 30 septembre 2000 17 h 01 GMT-07:00
```

101

Variable-length argument lists

J2SE 1.5 introduced language syntax for specifying a variable number of arguments ("varargs") to a method (think `printf` in C)

- Prior to this feature, methods had to be overloaded to take one, two, three, etc. arguments, or you had to pass in an array
- Now there is a special keyword `...` that indicates that there are multiple arguments
- The vararg is treated like an array in the method body
 - Varargs have a `length` and are zero-indexed
- A method can only have one variable-length argument list
 - Only the last argument to a method can have variable length
- The argument to `Arrays.asList` has variable arguments

```
List l = Arrays.asList("One", "Two", "Three");
```

102

Variable-length argument lists

An example of a variable-length argument list:

```
package edu.pdx.cs410J.j2se15;

public class VarArgs {

    private static void printSum(String header,
                                int... ints) {
        int sum = 0;
        for (int i : ints) {
            sum += i;
        }
        System.out.print(header);
        System.out.println(sum);
    }

    public static void main(String[] args) {
        printSum("1+2+3 = ", 1, 2, 3);
        printSum("1+2+3+4+5 = ", 1, 2, 3, 4, 5);
        printSum("2+4+6+8 = ", 2, 4, 6, 8);
    }
}
```

103

J2SE 1.5 text formatting

One of the deficiencies of Java's text formatting capabilities was that you had to invoke `print` (or `StringBuilder.append`) multiple times, or you had to create an `Object` array to pass to a `java.text.MessageFormat`'s `format` method

Variable-length argument lists allow the Java API to provide C-style `printf` and `scanf` behavior

- A `printf` method has been added to `java.io.PrintStream`
- Most of the formatting work is done by the `java.util.Formatter` class
- `Formatter` supports formatting the primitive types (`int`, etc.), `Strings`, `Calendars`, etc.
- A new method `String.format()` offers the functionality of `sprintf` (formatting to a `String`)
- The `format` is a superset of what is offered in C, but attempts to convert incompatible types (a `Calendar` to an `int`) will result in an exception being thrown

104

Format string syntax

The general form of the format string is:

`%[argument$][flags][width][.precision]conversion`

- The `argument` is the index of the argument in the `varargs` list
- `flags` are characters that modify the output format
- `width` is the minimum number of characters that should be written for the argument
- `precision` usually restricts the number of characters that should be written (dates and times do not have a precision)
- `conversion` is a character that indicates how the argument should be formatted

105

Formatting times

The `t` conversion character can be followed by one of the following (like POSIX `strftime`):

H	"Hour of day" (00 - 23)
I	"12-hour hour" (01 - 12)
k	"24-hour hour" (0 - 23)
l	"12-hour hour" (1 - 12)
M	"minute" (00 - 59)
S	"second" (00 - 60)
L	"millisecond" (000 - 999)
N	"nanosecond" (000000000 - 999999999)
p	am/pm
T	AM/PM
z	RFC 822 time zone offset (e.g. -0800)
Z	String time zone (PDT)
s	Seconds since epoch
E	Milliseconds since epoch

107

Format string syntax

This table summarizes the various conversion characters

b	"boolean" true or false
h	"hashcode (<code>arg.hashCode()</code> in hexadecimal)
s	"string" <code>toString</code> is invoked
c	"character"
d	"decimal integer"
o	"octal"
x	"hexadecimal"
e	"floating point" (in scientific notation)
f	"floating point"
g	"floating point" (scientific for large exponents)
a	"floating point" (significant and exponent)
t	"time" (date and time)
%	literal percent
n	"newline" (platform-specific line separator)

106

Formatting dates

The `t` conversion character can be followed by one of the following (like POSIX `strftime`):

B	"full month" (e.g. January)
b	"short month" (e.g. Jan)
A	"full day of week" (e.g. Sunday)
a	"short day of week" (e.g. Sun)
Y	"four-digit year"
y	"two-digit year"
j	"day of year"
m	"two-digit month"
d	"two-digit day of month"
e	"day of month" (one or two digits)

The following flags can be applied to format strings:

-	"left justified"
^	"upper case"
#	"alternate form"
+	numerics will always have a sign
	positive numerics have leading space
0	numerics are zero-padded
,	numerics have grouping separators
(negative numerics are enclosed in parentheses

108

An example of using formatting

```
package edu.pdx.cs410J.j2se15;

import java.io.PrintStream;
import java.util.Calendar;

public class Formatting {
    public static void main(String[] args) {
        PrintStream out = System.out;
        out.printf("%s\n", "Hello World");

        Calendar today = Calendar.getInstance();
        out.printf("Today's date is: %tm/%td/%tY\n",
            today, today, today);
        out.printf("The current time is: %tl:%tM %tp\n",
            today, today, today);

        out.printf("%f/%.2f = %f\n", 2.0, 3.0, (2.0/3.0))

        for (int i = 0; i < 3; i++) {
            out.printf("%5s%5s%5s\n", i, i+1, i+2);
        }

        out.printf("%-10s%s\n", "left", "right");
    }
}
```

Summary

Java's standard class libraries provide a vast array of functionality

- Basic language features: String, StringBuilder, Class, "wrapper" classes, Math
- Facilities for performing byte-based or character-based I/O: File, OutputStream, PrintStream, FileWriter, BufferedReader
- Handy utilities: Date, Calendar, BitSet, StringTokenizer
- Collection classes: Vector, List, Iterator, HashMap, Comparator