Comp-303 : Programming Techniques
Lecture 13

Alexandre Denault
Computer Science
McGill University
Winter 2004

Last lecture . . .

- Java Sockets work a lot like any other TCP/IP sockets.
- Java provides two socket objects : *Socket* and *ServerSocket*.
- Communicating over sockets is identical to reading/writing to files.
- Java provides serialization as a mean to save object and transmit them.
- *ObjectOutputStream* and *ObjectInputStream* can be used file any kind of streams (files, socket, etc).
- By implementing *Serializable*, your objects will be serializable.

Today's lecture

- A small change to Assignment 2.
- Corrections / Confusions in the previous lectures
  - Deterministic / Determined
  - String equality
  - Public, protected, private
  - Socket behavior
- Using JavaDoc for automatic documentation generation
- Review for midterm

Assignment 2

- Because I didn't figure in associativity correctly when planning the *SumOfSquare* Adder, the specification in the assignment won't calculate the correct result.
- As such, I'm removing the *SumOfSquare* Adder from the assignment.
- The rest of the assignment remains unchanged.
- That means less work for you.

Deterministic / Determined

- *Determined*: a procedure is considered determined if the acceptable output is unique.
- *Undetermined*: a procedure is considered undetermined if the acceptable output is a set.
- *Deterministic Implementation*: these procedures will produce exactly the same output when give the same input.
- *Non-deterministic Implementation*: these procedures can produce different outputs when give the same input.

Comparing Deterministic and Determined

	Determined	Undetermined
Deterministic	The procedure always returns the same value	The procedure always returns the same output for a given input.
Non-deterministic	By definition, cannot exists	The procedure can return any given set of output for any given set of input.

Comparing Strings

- For optimization purposes, strings in Java are handled in two ways.
- Unfortunately, comparing the two types of strings can yield different results.
- Some strings are declared as objects.

```
String objectString = new String("test");
```

- Others are declared like a primitive. They are store as constants in the class file.

```
String constantString = "test";
```

- The tricky thing is that these two types of string react differently to the == sign.

Proof by example

```
String constantString = "test";
String objectString = new String("test");

System.out.println("test" == "test");
System.out.println(constantString == "test");
System.out.println(objectString == "test");
System.out.println(objectString == new String("test"));
System.out.println(objectString == constantString);

System.out.println(constantString.equals("test"));
System.out.println(objectString.equals("test"));
System.out.println(objectString.equals(new String("test")));
System.out.println(objectString.equals(constantString));
System.out.println(constantString.equals(constantString));
```

Proof by example : Results

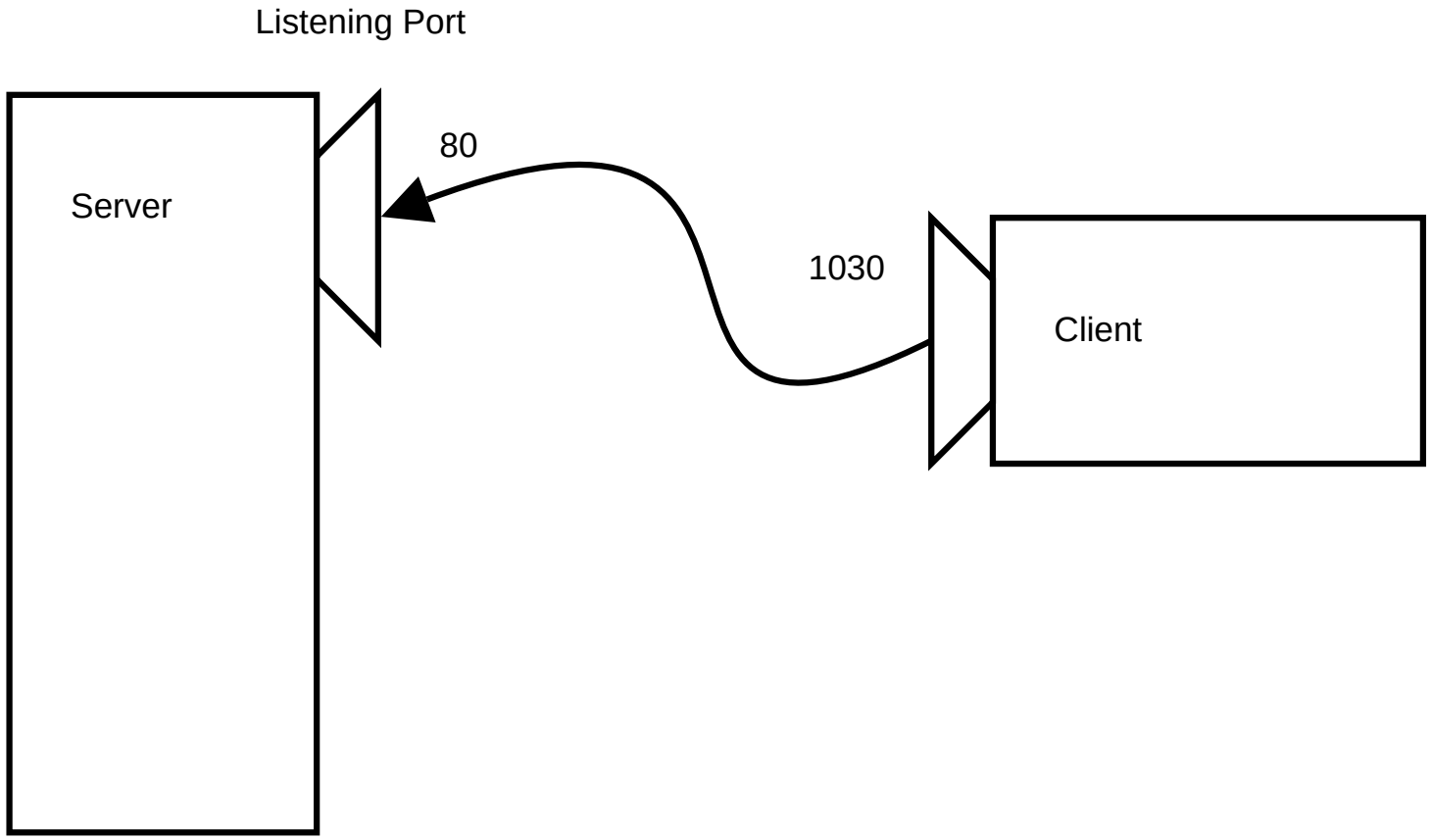
```
String constantString = "test";
String objectString = new String("test");

System.out.println("test" == "test"); \\ true
System.out.println(constantString == "test"); \\ true
System.out.println(objectString == "test"); \\ false
System.out.println(objectString == new String("test")); \\ false
System.out.println(objectString == constantString); \\ false

System.out.println(constantString.equals("test")); \\ true
System.out.println(objectString.equals("test")); \\ true
System.out.println(objectString.equals(new String("test"))); \\ true
System.out.println(objectString.equals(constantString)); \\ true
System.out.println(constantString.equals(constantString)); \\ true
```

In other words, equals has constant behavior. When comparing strings, using the *equals* method.

Public/Protected/Private

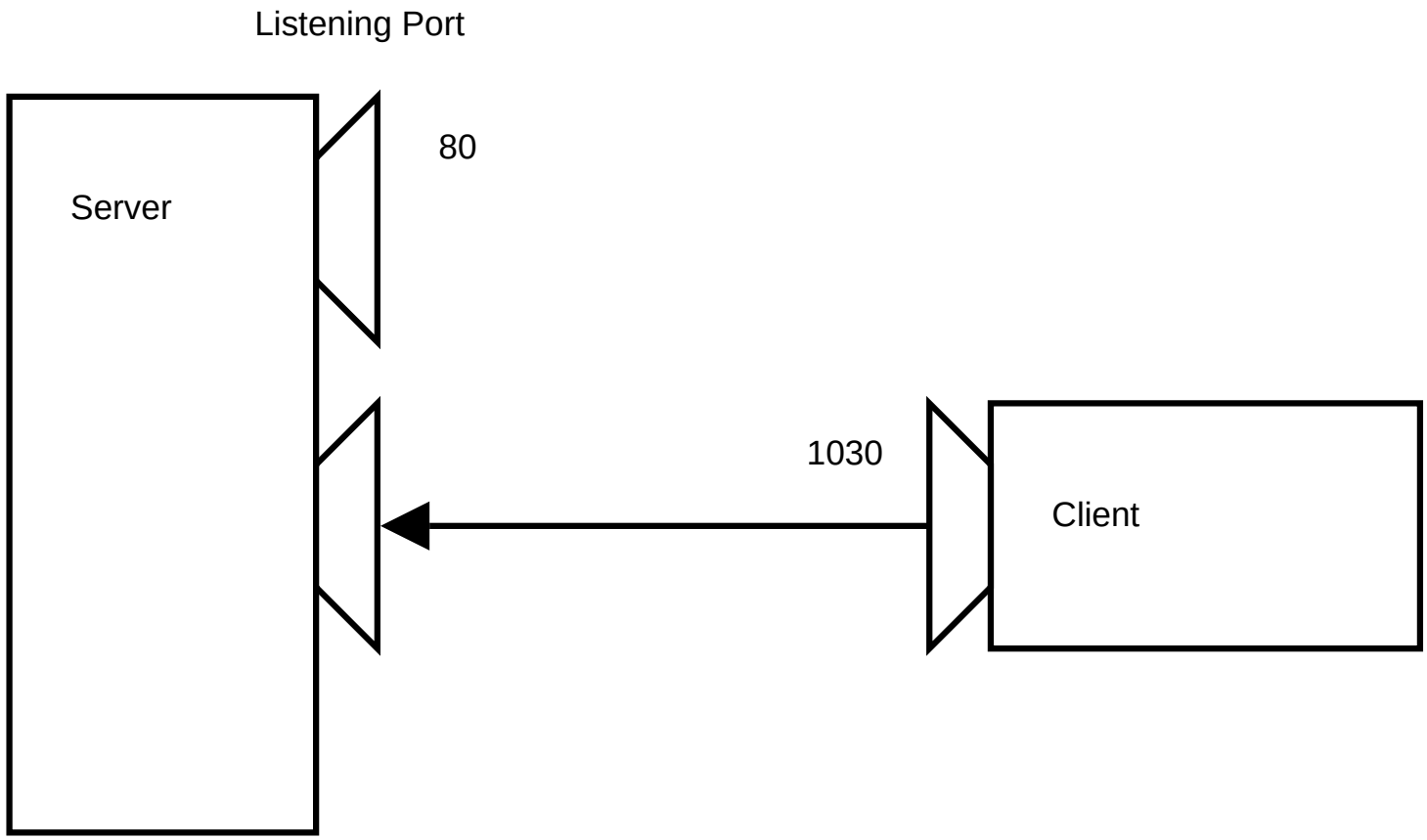
Java	Interfaces, Classes	Variables, Methods, Inner-Classes
Public	visible everywhere	visible everywhere, inherited by all subclasses
Protected	-	visible to any class in the package, inherited by all subclasses in or outside the package
blank	visible only in current package	visible to any class in the package, inherited by all subclasses in the package but not outside package
Private	-	not visible to any class, not inherited by any class

Client requests connection



To make a connection request, the client tries to rendezvous with the server on the server's machine and port.

Server accepts connection



Upon acceptance, the server gets a new socket.

JavaDoc

- Sun has a different approach to documentation.
- The idea is that the comments in the code should be the documentation.
- The Java API is built upon this idea.
- JavaDoc is the tool used to extract comments from code and build the documentation.
- To properly format the documentation, you need to add a few tags to your comments.
- Comments written for JavaDoc can be formatted using HTML.

Writing API Specifications

- API Specifications are defined by the documentation comments in the source code and any documents marked as specifications reachable from those comments.
- API Specifications are a contract between callers and implementation.
- API Specifications assertions need to be implementation-independent.

The format of a comment

```
/**
* Returns an Image object that can then be painted on the screen.
* The url argument must specify an absolute {@link URL}. The name
* argument is a specifier that is relative to the url argument.
* <p>
* This method always returns immediately, whether or not the
* image exists. When this applet attempts to draw the image on
* the screen, the data will be loaded. The graphics primitives
* that draw the image will incrementally paint on the screen.
*
* @param  url  an absolute URL giving the base location of the image
*         name the location of the image, relative to the url argument
* @return      the image at the specified URL
* @see         Image
*/
public Image getImage(URL url, String name) {
```

Proper documentation

- The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the API item.
- This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag.
- As much as possible, write doc comments as an implementation-independent API specification.
- Define clearly what is required and what is allowed to vary across platforms/implementations.
- Make it complete enough for conforming implementors. Realistically, include enough description so that someone reading the source code can write a substantial suite of conformance tests.

Proper documentation (cont.)

- Where appropriate, mention what the specification leaves unspecified or allows to vary among implementations.
- If you must document implementation-specific behavior, please document it in a separate paragraph with a lead-in phrase that makes it clear it is implementation-specific.
- If the implementation varies according to platform, then specify "On ¡platform¿" at the start of the paragraph.
- Javadoc tool duplicates (inherits) or links comments for methods that override or implement other methods.
- So lines won't wrap, limit any doc-comment lines to 80 characters.

Tags

- @author (classes and interfaces only, required)
- @version (classes and interfaces only, required) (see footnote 1)
- @param (methods and constructors only)
- @return (methods only)
- @exception (@throws is a synonym added in Javadoc 1.2)
- @see
- @since
- @serial (or @serialField or @serialData)
- @deprecated (see How and When To Deprecate APIs)

Note: The order is important. Try to use tags in the following order.

@author

- You can provide one *@author* tag, multiple @author tags, or no *@author* tags.
- Multiple *@author* tags should be listed in chronological order, with the creator of the class listed at the top.
- If the author is unknown, use *unascribed* as the argument to *@author*.
- he @author tag is not critical, because it is not included when generating the API specification, and so it is seen only by those viewing the source code.

@version

- Useful when multiple version of a class is available.
- Follows this format : 1.39, 02/28/97 (mm/dd/yy)

@param

- The *@param* tag is followed by the name (not data type) of the parameter, followed by a description of the parameter.
- By convention, the first noun in the description is the data type of the parameter.
- An exception is made for the primitive int, where the data type is usually omitted.
- Parameter names are lowercase by convention.

@return

- The *@return* is use to describe the output of a method.
- Omit *@return* for methods that return void and for constructors.
- Include it for all other methods, even if its content is entirely redundant with the method description.
- Whenever possible, supply return values for special cases (out-of-bounds).

@deprecated

- The *@deprecated* description in the first sentence should at least tell the user when the API was deprecated and what to use as a replacement.
- Only the first sentence will appear in the summary section and index.
- If the member has no replacement, the argument to @deprecated should be *No replacement.*

```
/**
* @deprecated  As of JDK 1.1, replaced by {@link #setBounds(int,int,int,int)}
*/
```

@since

- Specify the product version when the name was added to the API specification.
- The Javadoc tool does not proliferate the value of the *@since* clause down the hierarchy.
- When a class (or interface) is introduced, specify one *@since* tag in its class description and no *@since* tags in the members.

@throws

- A *@throws* tag should be included for any checked exceptions (declared in the throws clause)
- They should also be included for any unchecked exceptions that the caller might reasonably want to catch, with the exception of NullPointerException.
- Errors should not be documented as they are unpredictable.

@see

- The *@see* tag is a reference to other packages / classes / methods / constructors / fields.
- They should be ordered from nearest to farthest access, from least-qualified to fully-qualified.

```
@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#method(Type, Type,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package
```

@serial, @serialField, @serialData

- The following tags are used to document custom serialization behavior.

{@link}

- The style guide encourages programmers to add links for API names (package, classes, methods, . . .) using the {*@link*} tag.
- A link to an API name should be added if:
  - The user might actually want to click on it for more information (in your judgment).
  - Only for the first occurrence of each API name in the doc comment (don't bother repeating a link).

Comment template

```
package java.blah;

import java.blah.blahdy.BlahBlah;

/**
* OVERVIEW: Class description goes here.
*
* @version  1.82 18 Mar 1999
* @author  Firstname Lastname
*/

public class Blah extends SomeClass {
   /* A class implementation comment can go here. */
```

Comment template (cont.)

```
/** classVar1 documentation comment */
public static int classVar1;

/**
 * classVar2 documentation comment that happens to be
 * more than one line long
 */
private static Object classVar2;

/** instanceVar1 documentation comment */
public Object instanceVar1;

/** instanceVar2 documentation comment */
protected int instanceVar2;

/** instanceVar3 documentation comment */
private Object[] instanceVar3;
```

Comment template (cont.)

```
/**
 * ...constructor Blah documentation comment...
 */
public Blah() {
   // ...implementation goes here...
}

/**
 * ...method doSomething documentation comment...
 *
 * <p> REQUIRES: blah blah
 * <p> MODIFIES: blah blah
 * <p> EFFECTS:  blah blah
 * <p>
 */
public void doSomething() {
   // ...implementation goes here...
}
```

Comment template (cont.)

```
/**
 * ...method doSomethingElse documentation comment...
 *
 * <p> REQUIRES: blah blah
 * <p> MODIFIES: blah blah
 * <p> EFFECTS:  blah blah
 * <p>
 * @param  someParam description
 * @return description
 */
public int doSomethingElse(Object someParam) {
   // ...implementation goes here...
}

}
```

Lecture 1: Programming Techniques

- Decomposition
- Abstraction
- Abstraction in modern programming language
- Four different types of abstraction.

Lecture 2: How does Java work?

- Definition of Object-oriented programming and why use it
- Abstract classes and Interfaces
- Overloading and overridding

Lecture 3: Objects, Types and Variables

- Objects and primitives
- Assignment
- Mutability
- Type Substitutability / Checking
- Type Conversion / Casting
- Method dispatch
- Packages

Lecture 4: Crash course in UML

- UML
- Class diagrams
- Sequence diagrams

Lecture 5: Procedural Abstraction

- Procedural abstraction
- Requires / Modifies / Effects clauses
- Implicit Inputs
- Properties of procedures

Lecture 6: Data Abstraction

- Data Abstraction
- Abstract Data Types
- Instance variables
- Records
- Methods inherited from Object

Lecture 7: Exceptions

- Exceptions
- Building Exceptions
- Throwing Exceptions
- Handling Exceptions
- Defensive programming

Lecture 8: Iteration Abstraction

- Iteration
- Iterators
- Generators
- Iterators over iterators
- Functions as objects (plug-ins)

Lecture 9: Type Hierarchy

- Type Hierarchy
- Method Dispatch
- How to use protected
- Rules to change specification

Lecture 10 : Polymorphic Abstractions

- Using Polymorphic Abstractions
- Equality
- Comparing
- Containers
- Flexibility by using interfaces (Adder)

Lecture 11: Thread Programming

- Creating Threads
- Life of a Thread
- Priority / Scheduling in threads
- Synchronizing / Locking
- Dangers of threading

Lecture 12: Socket and Serialization

- Creating Sockets (client/server)
- Reading and writing to sockets
- Serialization

Summary

- Corrections / Confusions in the previous lectures
  - Deterministic / Determined
  - String equality
  - Public, protected, private
  - Socket behavior
- Using JavaDoc for automatic documentation generation
- Review for midterm

Tool of the day: JavaDoc

- I think I've talked enough about JavaDoc for today.