Object Oriented Programming

1. Testing
2. Debugging
3. Introduction to Java I/O

Functional Testing

- *Software testing*: the process used to help identify the correctness, completeness, security and quality of developed computer software
- Goal of *functional testing*: determine system meets customer's specifications.
- *Black box testing*:
  - Test designer ignores internal structure of implementation.
  - Test driven by expected external behavior of the system
  - System is treated as a "black box": behavior can be observed, but internal structure is unknown.

Test Design, Plan and Test Cases

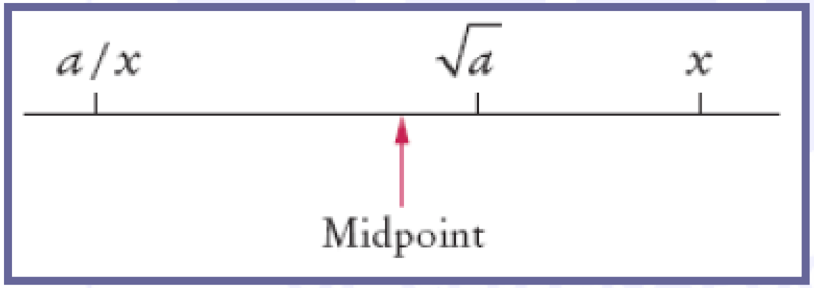
- Test design generally begins with an analysis of
  - Functional specifications of system, and
  - Use cases: ways in which the system will be used.
- A test case is defined by
  - Statement of case objectives;
  - Data set for the case;
  - Expected results.
- A test plan is a set of test cases. To develop it:
  - Analyze feature to identify test cases.
  - Consider set of possible states object can assume.
  - Tests must be representative.

Unit Tests

- The single most important testing tool
- Checks a single method or a set of cooperating methods
- You don't test the complete program that you are developing; you test the classes in isolation
- For each test, you provide a simple class called a *test harness*
- Test harness feeds parameters to the methods being tested

Example: Setting Up Test Harnesses

- To compute the square root of *a* use a common algorithm:
  1. Guess a value *x* that might be somewhat close to the desired square root (*x* = *a* is ok)
  2. Actual square root lies between *x* and *a/x*
  3. Take midpoint (*x* + *a*/*x*) / 2 as a better guess



  4. Repeat the procedure. Stop when two successive approximations are very close to each other
- The method converges rapidly. Demo: root1
  - There are 8 guesses for the square root of 100:

Testing the Program

- Does the `RootApproximator` class work correctly for all inputs?
  It needs to be tested with more values
- Re-testing with other values repetitively is not a good idea; the tests are not repeatable
- If a problem is fixed and re-testing is needed, you would need to remember your inputs
- Solution: Write test harnesses that make it easy to repeat unit tests

Providing Test Input

- There are various mechanisms for providing test cases
- One mechanism is to hardwire test inputs into the test harness.
  - Example: root2/RootAproximationHarness1
- Simply execute the test harness whenever you fix a bug in the class that is being tested
- Alternative: place inputs on a file instead

Providing Test Input

- You can also generate test cases automatically
- For few possible inputs, feasible to run through a (representative) number of them with a loop
  - Example: root2/RootAproximationHarness2
- Previous test restricted to small subset of values
- Alternative: random generation of test cases
  - Example: root2/RootAproximationHarness3

Providing Test Input

- Selecting good test cases is an important skill for debugging programs
- Test all features of the methods that you are testing
- Test *typical test cases*
  100, 1/4, 0.01, 2, 10E12, for the **`SquareRootApproximator`**
- Test *boundary test cases*: test cases that are at the boundary of acceptable inputs
  0, for the **`SquareRootApproximator`**

Providing Test Input

- Programmers often make mistakes dealing with boundary conditions
  - Division by zero, extracting characters from empty strings, and accessing null pointers
- Gather negative test cases: inputs that you expect program to reject
  - Example: square root of -2. Test passes if harness terminates with assertion failure (if assertion checking is enabled)

Reading Test Inputs From a File

- More elegant to place test values in a file
- Input redirection: `java Program < data.txt`
- Some IDEs do not support input redirection. Then, use command window (shell).
- Output redirection: `java Program > output.txt`
- Example: root2/RootAproximationHarness4
- File test.in:

```
100
4
2
1
0.25
0.01
```

- Run the program:

```
java RootApproximatorHarness4 < test.in > test.out
```

Test Case Evaluation

- How do you know whether the output is correct?
- Calculate correct values by hand
  - E.g., for a payroll program, compute taxes manually
- Supply test inputs for which you know the answer
  - E.g., square root of 4 is 2 and square root of 100 is 10
- Verify that the output values fulfill certain properties
  - E.g., square root squared = original value
- Use an Oracle: a slow but reliable method to compute a result for testing purposes
  - E.g., use `Math.pow` to slower calculate $x^{1/2}$ (equivalent to the square root of x)
- Example: root3/RootAproximationHarness5, 6

Regression Testing

- Save test cases
- Use saved test cases in subsequent versions
- A test suite is a set of tests for repeated testing
- Cycling = bug that is fixed but reappears in later versions
- Regression testing: repeating previous tests to ensure that known failures of prior versions do not appear in new versions

Test Coverage

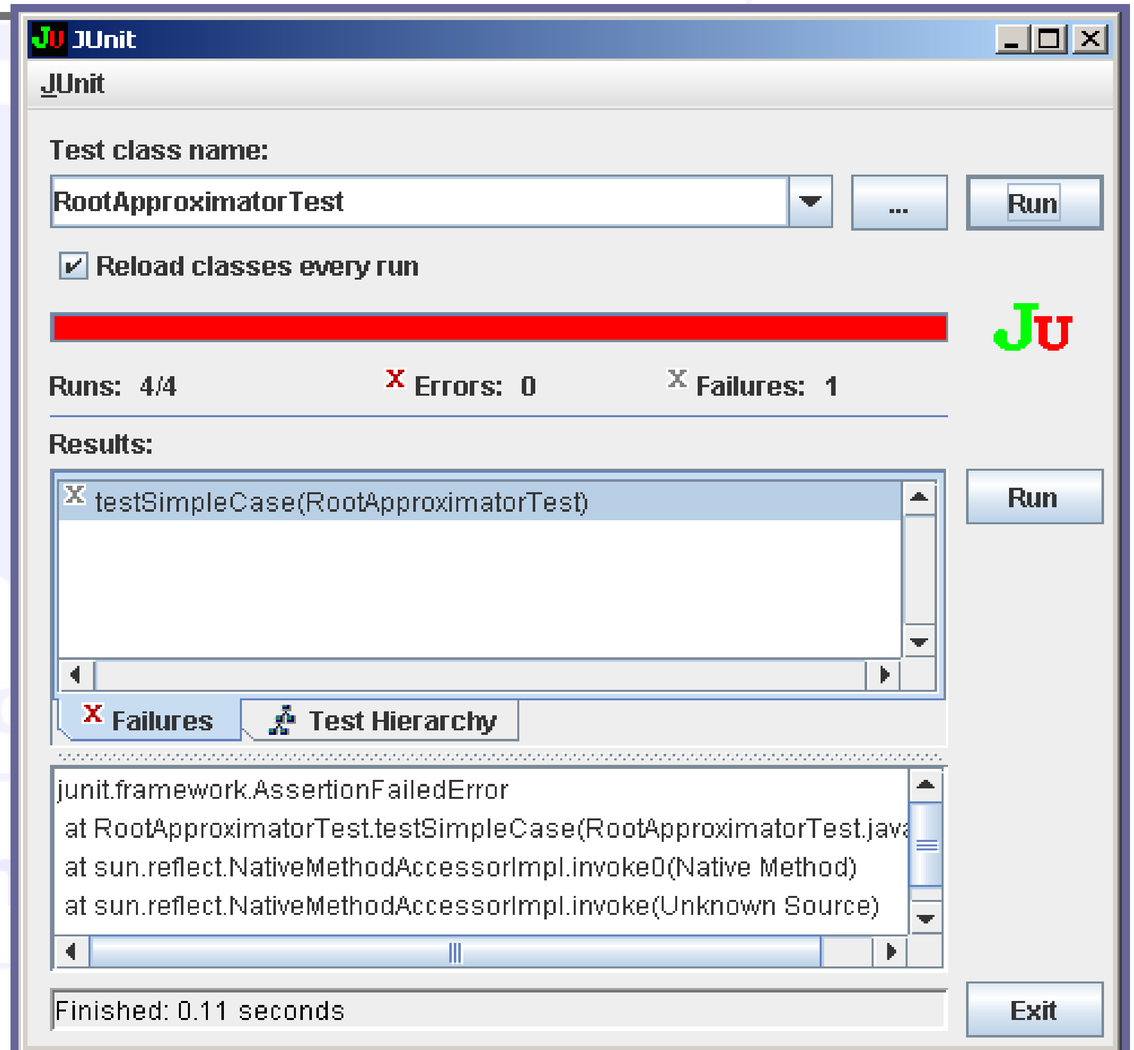
- Black-box testing: test functionality without consideration of internal structure of implementation
- White-box testing: take internal structure into account when designing tests
- Test coverage: measure of how many parts of a program have been tested
- Make sure that each part of your program is exercised at least once by one test case
  - E.g., make sure to execute each branch in at least one test case

Test Coverage

- Tip: write first test cases before program is written completely → gives insight into what program should do
- Modern programs can be challenging to test
  - Graphical user interfaces (use of mouse)
  - Network connections (delay and failures)
  - There are tools to automate testing in these scenarios
  - Basic principles of regression testing and complete coverage still hold

Unit Testing With JUnit

- http://junit.org
- Built into some IDEs like BlueJ and Eclipse
- Philosophy: whenever you implement a class, also make a companion test class
- Demo: project under junit subdirectory
- On the right there is a Swing UI example of junit 3.8.1
  - newer junit 4.0 is different, no Swing UI



Program Trace

- Messages that show the path of execution

```
if (status == SINGLE)
{
    System.out.println("status is SINGLE");
    . . .
}
```

- Drawback: Need to remove them when testing is complete, stick them back in when another error is found
- Solution: use the `Logger` class to turn off the trace messages without removing them from the program `(java.util.logging)`

Logging

- Logging messages can be deactivated when testing is complete
- Use global object `Logger.global`
- Log a message
- By default, logged messages are printed. Turn them off with

```
Logger.global.setLevel(Level.OFF);
```

- Logging can be a hassle (should not log too much nor too little)
- Some programmers prefer debugging (next section) to logging

```
Logger.global.info("status is SINGLE");
```

Logging

- When tracing execution flow, the most important events are entering and exiting a method
- At the beginning of a method, print out the parameters:

```
public TaxReturn(double anIncome, int aStatus) {
   Logger.global.info("Parameters: anIncome = " + anIncome
      + " aStatus = " + aStatus);
   . . .
}
```

- At the end of a method, print out the return value:

```
public double getTax() {
   . . .
   Logger.global.info("Return value = " + tax);
   return tax;
}
```

Logging

- The logging library has a set of predefined logging levels:

SEVERE	*The highest value;* intended for extremely important messages (e.g. fatal program errors).
WARNING	Intended for warning messages.
INFO	Informational runtime messages.
CONFIG	Informational messages about configuration settings/setup.
FINE	Used for greater detail, when debugging/diagnosing problems.
FINER	Even greater detail.
FINEST	*The lowest value;* greatest detail.

- In addition to these levels:
  - level ALL, which enables logging of all records, and
  - level OFF that can be used to turn off logging.
  - It is also possible to define custom levels (see Java SDK docs)
- Demo: loggingEx

Logging Benefits

- Logging can generate detailed information about the operation of an application.
- Once added to an application, logging requires no human intervention.
- Application logs can be saved and studied at a later time.
- If sufficiently detailed and properly formatted, application logs can provide audit trails.
- By capturing errors that may not be reported to users, logging can help support staff with troubleshooting.
- By capturing very detailed and programmer-specified messages, logging can help programmers with debugging.
- Logging can be a debugging tool where debuggers are not available, which is often the case with multi-threaded or distributed applications.
- Logging stays with the application and can be used anytime the application is run.

Logging Costs

- Logging adds runtime overhead, from generating log messages and from device I/O.
- Logging adds programming overhead, because extra code has to be written to generate the log messages.
- Logging increases the size of code.
- If logs are too verbose or badly formatted, extracting information from them can be difficult.
- Logging statements can decrease the legibility of code.
- If log messages are not maintained with the surrounding code, they can cause confusion and can become a maintenance issue.
- If not added during initial development, adding logging can require a lot of work modifying code.

Debugging

Using a Debugger

- Debugger = program to run your program and analyze its run-time behavior
- A debugger lets you stop and restart your program, see contents of variables, and step through it
- The larger your programs, the harder to debug them simply by logging
- Debuggers can be part of your IDE (Eclipse, BlueJ) or separate programs (JSwat)
- Three key concepts:
  - Breakpoints
  - Single-stepping
  - Inspecting variables

About Debuggers

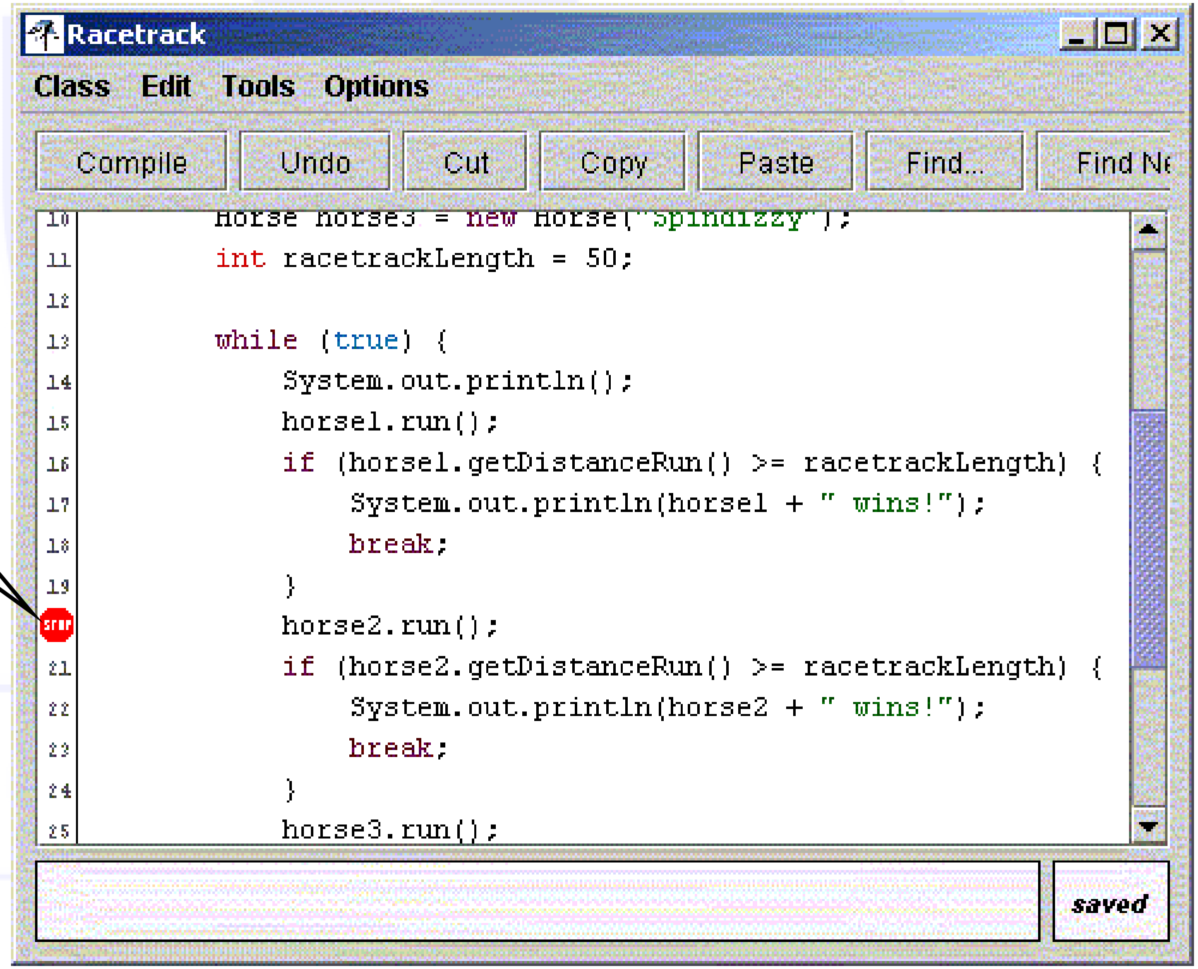
- Bugs happen
- Sometimes you can figure the problem out right away
- Sometimes you have to track down the problem
- A Debugger can be a big help
  - Sometimes it's exactly the tool you need
  - Sometimes it isn't
- Debuggers are all basically pretty much alike
  - "If you know one, you know them all"
- BlueJ comes with a nice, simple debugger that gives you all the most important functionality
  - Unfortunately, BlueJ's debugger is crash-prone

What a debugger does

- A debugger lets you step through your code line by line, statement by statement
- At each step you can examine the values of all your variables
- You can set breakpoints, and tell the debugger to just “continue” (run full speed ahead) until it reaches the next breakpoint
  - At the next breakpoint, you can resume stepping
- Breakpoints stay active until you remove them
- Execution is suspended whenever a breakpoint is reached
- In a debugger, a program runs at full speed until it reaches a breakpoint
- When execution stops you can:
  - Inspect variables
  - Step through the program a line at a time
  - Or, continue running the program at full speed until it reaches the next breakpoint

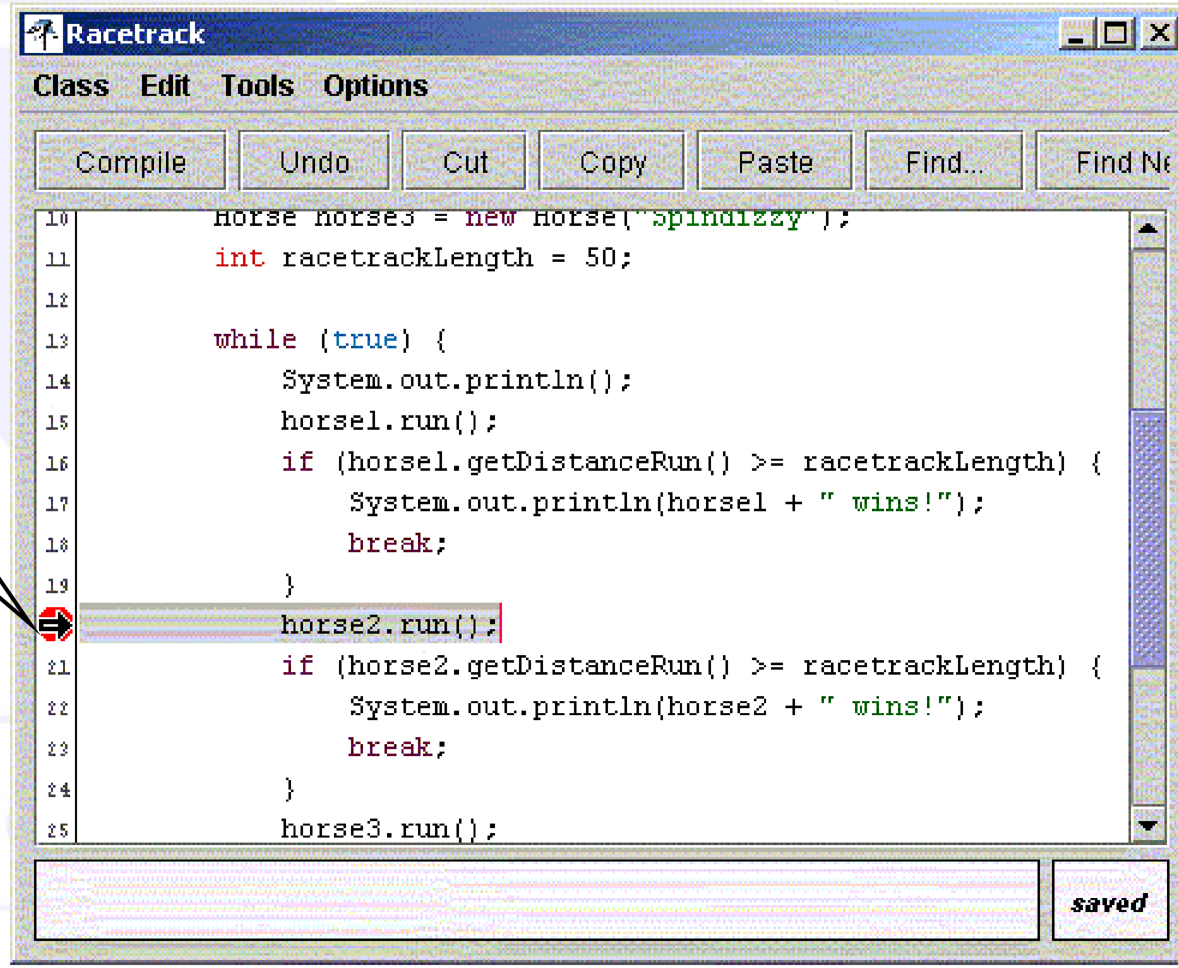
Setting a breakpoint

Click in the left margin to set or remove breakpoints



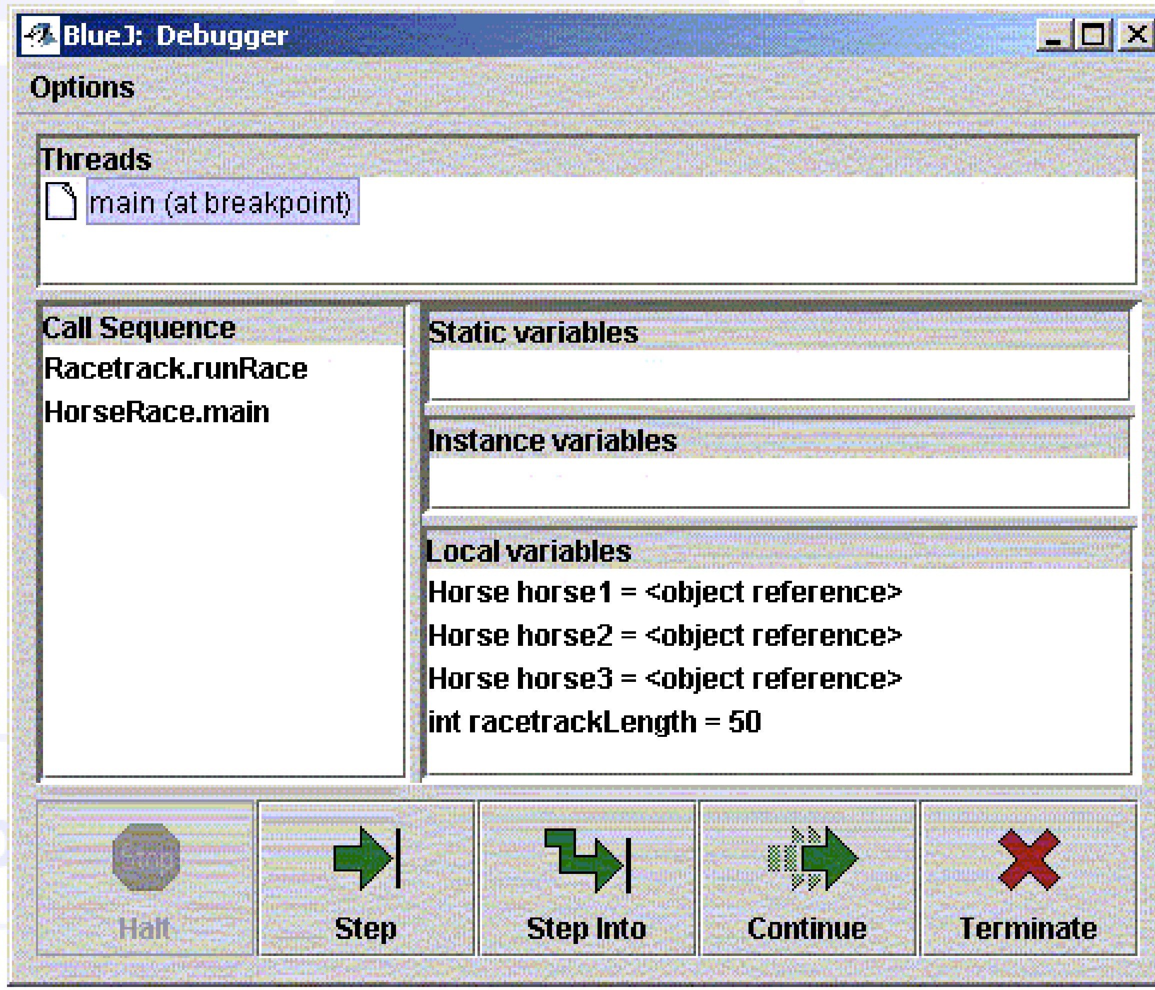
Reaching a breakpoint

Run your program normally; it will automatically stop at each breakpoint



The Debugger window

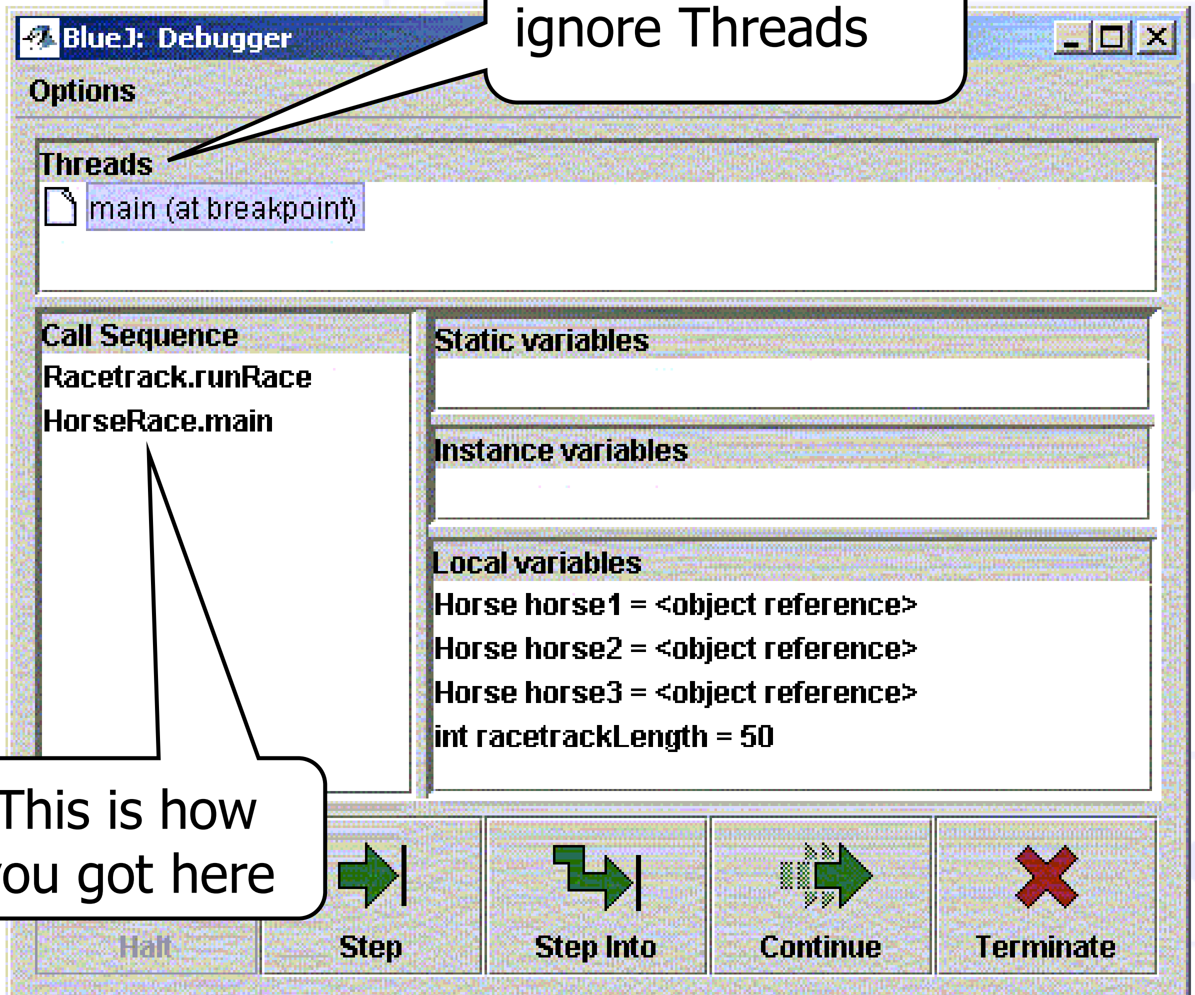
- When you reach a breakpoint, the debugger window will open
- Or, you can open it yourself from the menus



Parts of the Debugger window

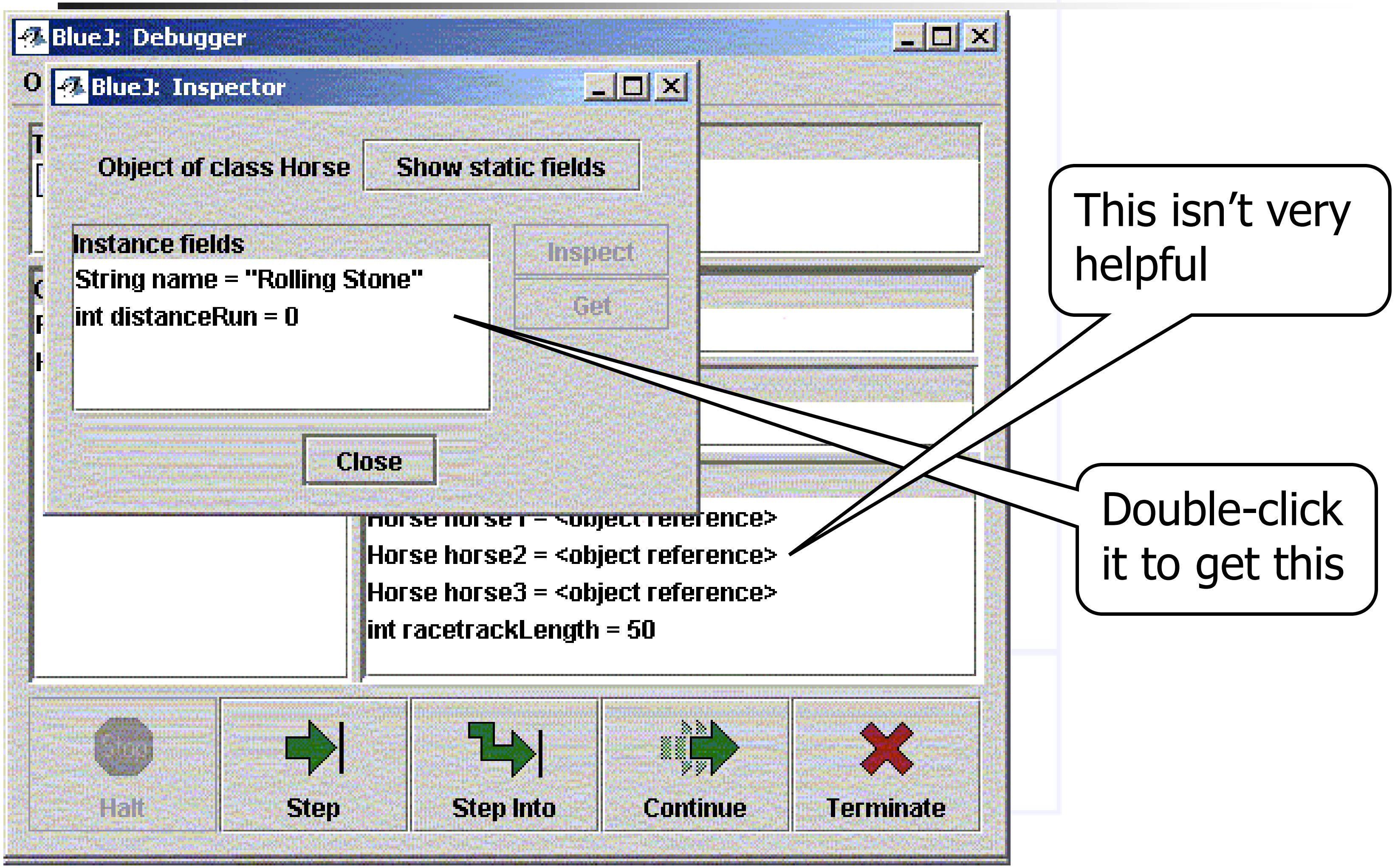
You can usually ignore Threads

Your variables and their values, as it says



This is how you got here

Viewing objects



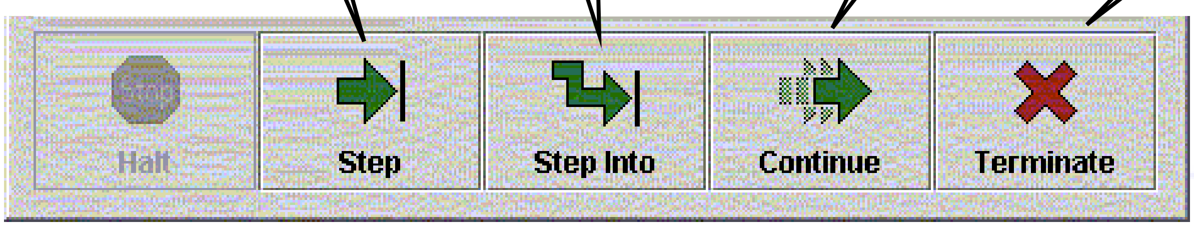
Debugger commands

Step ahead one statement

Step ahead one statement; but if it calls a method, step into the method

Quit debugging and just run, until you hit another breakpoint

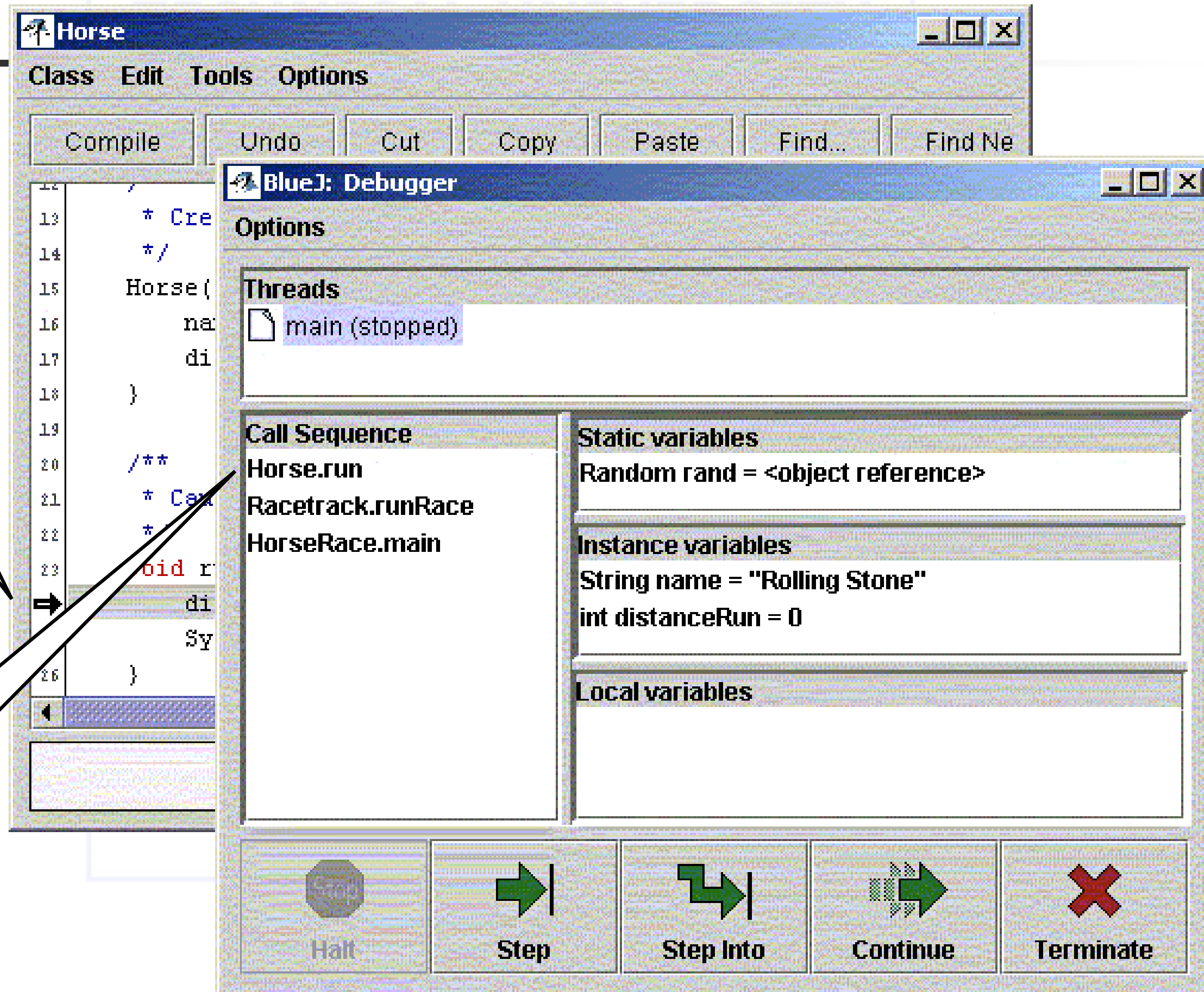
Quit



Stepping into a method

Here we are in the new method

This is how we got here



Cautions

- The BlueJ debugger is *very* flaky—use with care
- Debuggers can be very helpful, but they are no substitute for thinking about the problem
- Often, *print statements* are still the better solution

Introduction to Java I/O

Introduction to Java I/O

- The Java I/O system is very complex
  - It tries to do many different things with re-usable components
  - There are really *three* I/O systems, an original system release in JDK 1.0, and a newer system in released in JDK 1.2 that overlays and partially replaces it
  - The **`java.nio`** package from JDK 1.4 is even newer
- Performing I/O requires a programmer to use a series of complex classes
  - Convenience classes such as **`StdIn`**, **`FileIn`**, and **`FileOut`** are usually created to hide this complexity

Introduction to Java I/O (`java.io`)

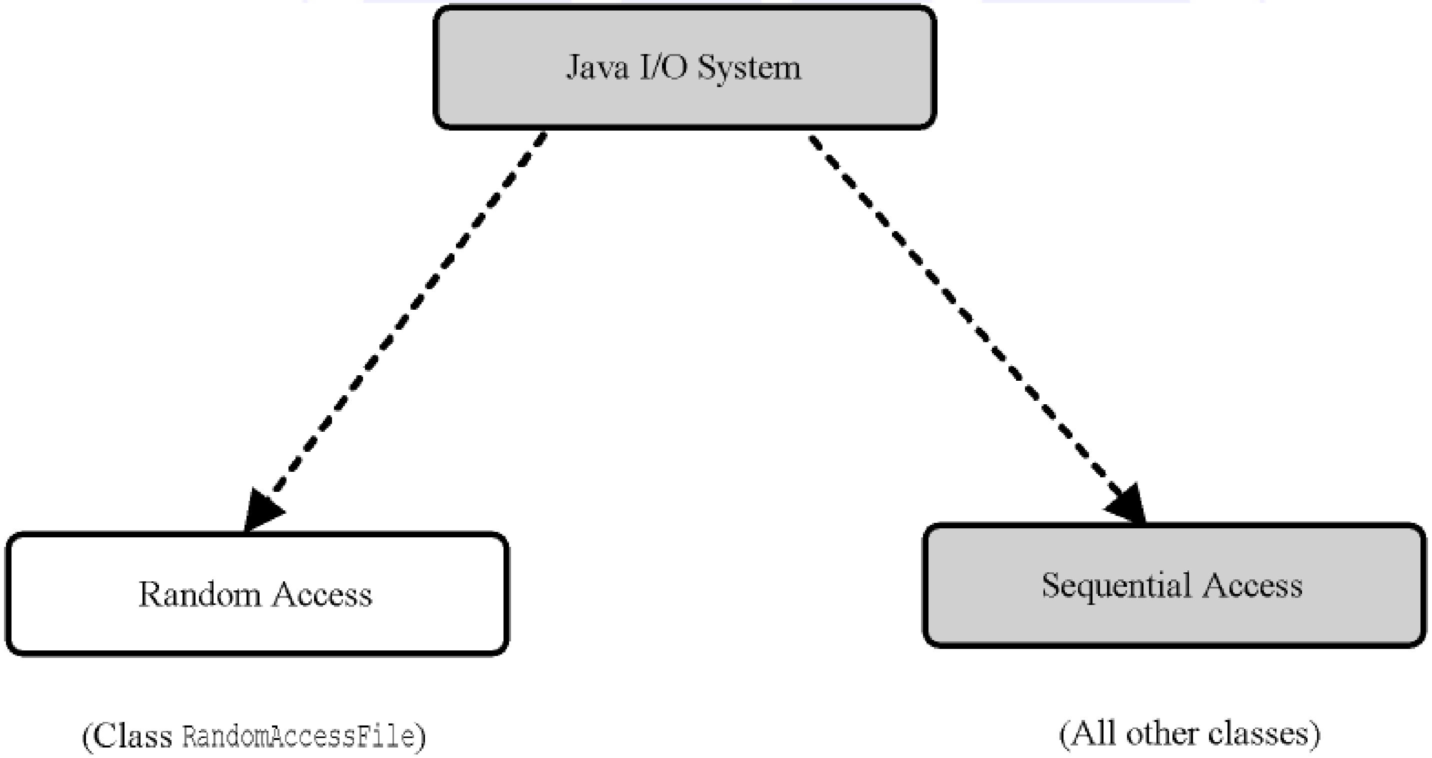
- Reasons for Java I/O complexity:
  - Many different types of sources and sinks
  - Two different types of file access
    - Sequential access
    - Random access
  - Two different types of storage formats
    - Formatted
    - Unformatted
  - Three different I/O systems (old and new)
  - A variety of “filter” or “modifier” classes

Random Access vs. Sequential Access

- Sequential access
  - A file is processed a byte at a time
  - It can be inefficient
- Random access
  - Allows access at arbitrary locations in the file
  - Only disk files support random access
    - `System.in` and `System.out` do not
  - Each disk file has a special file pointer position
    - You can read or write at the position where the pointer is

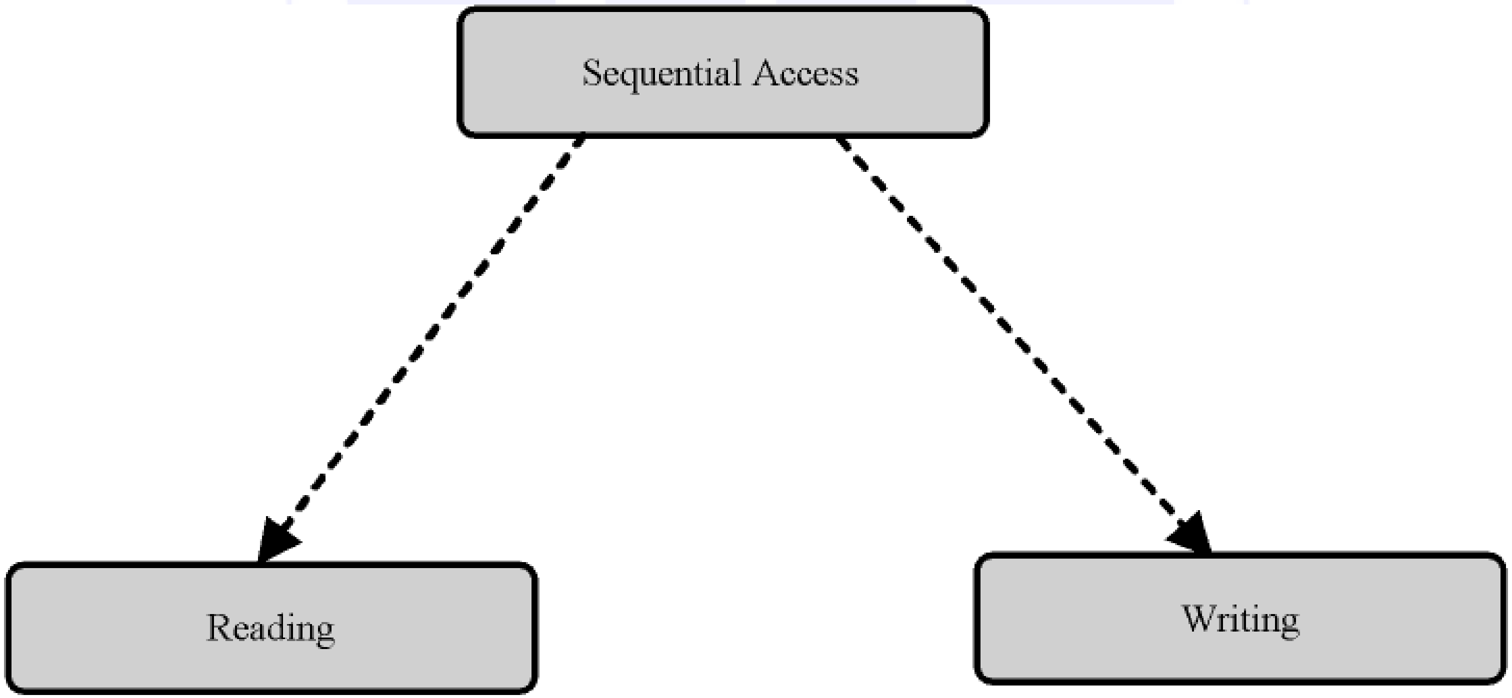
Structure of the Java I/O System (`java.io`)

- The Java I/O System is divided into sequential and random access classes:

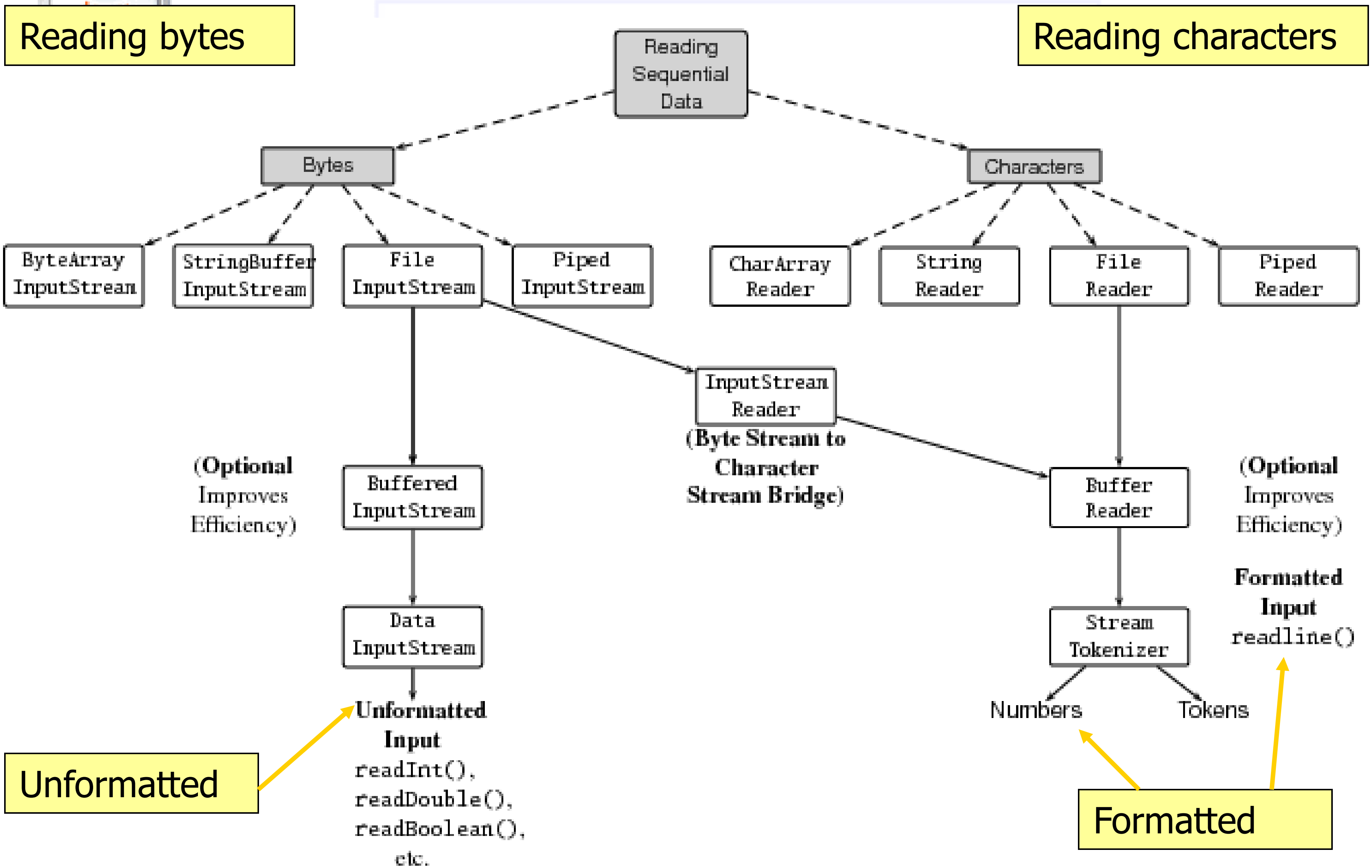


Structure of the Java I/O System (`java.io`)

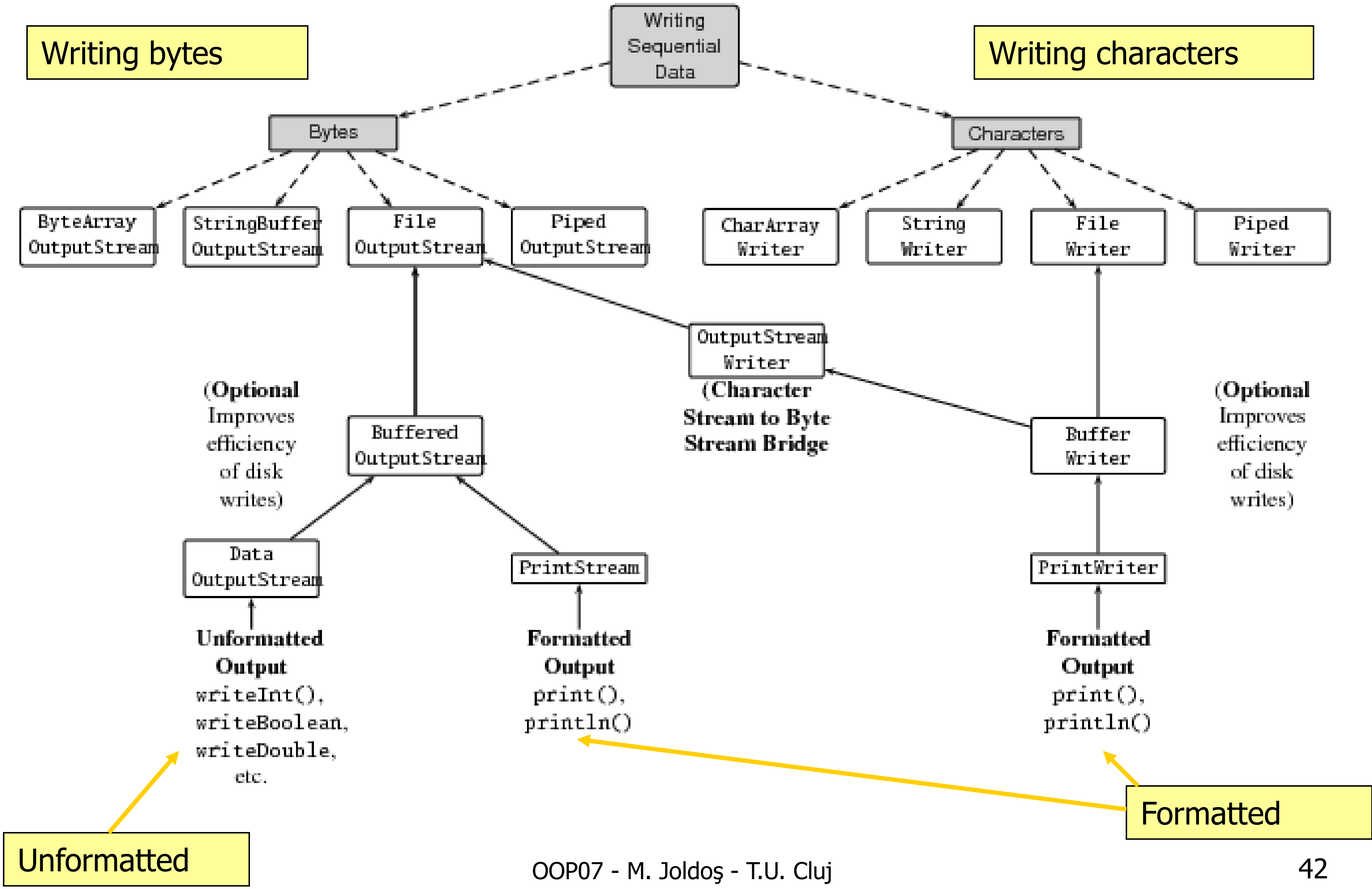
- Sequential access is further divided into classes for reading and classes for writing:



Classes for Reading Sequential Data in `java.io`



Classes for Writing Sequential Data in `java.io`



Exceptions

- All Java I/O classes throw exceptions, such as the `FileNotFoundException` and the more general `IOException`
- Java programs must explicitly trap I/O exceptions in a `try` / `catch` structure to handle I/O problems.
  - This structure *must* handle `IOException`, which is the general I/O exception class
  - It may handle lower level exceptions separately, such as the `FileNotFoundException`. This allows the program to offer the user intelligent information and options in the event that a file is not found

Using Java I/O

- The general procedure for using Java I/O is:
  - Create a `try/catch` structure for I/O exceptions
  - Pick an input or output class based on the type of I/O (formatted or unformatted, sequential or direct) and the type of input or output stream (file, pipe, etc.)
  - Wrap the input or output class in a buffer class to improve efficiency
  - Use filter or modifier classes to translate the data into the proper form for input or output (e.g., `DataInputStream` or `DataOutputStream`)

Example: Reading `String`s from a Formatted Sequential File

- Select a `FileReader` class to read formatted sequential data.
  - Open the file by creating a `FileReader` object
  - Wrap the `FileReader` in a `BufferedReader` for efficiency
  - Read the file with the `BufferedReader` method `readLine()`.
  - Close file with the `FileReader` method `close()`.
  - Handle I/O exceptions with a `try/catch` structure

Example

Enclose I/O in a `try/catch` structure

Open file by creating a `FileReader` wrapped in a `BufferedReader`

Read lines with `readLine()`

Close file with `close()`

Handle exceptions

```
// Trap exceptions if they occur
try
{
  // Create BufferedReader
  BufferedReader in = null;
  in =  new BufferedReader(  new FileReader(args[0]) );
 // Read file and display data
  while( (s = in.readLine()) != null)
 {
    System.out.println(s);
  }
}
// Catch FileNotFoundException
catch (FileNotFoundException e)
{
  System.out.println("File not found: " + args[0]);
}
// Catch other IOExceptions
finally
{
 // Close file
  if (in != null) in.close();
}
```

Scanners

- Instead of reading directly from `System.in`, or a text file use a `Scanner`
  - Always have to tell the `Scanner` what to read
  - E.g. instantiate it with a reference to read from `System.in`

```
java.util.Scanner scanner =
    new java.util.Scanner(System.in);
```

- What does the `Scanner` do?
  - Breaks down input into manageable units, called *tokens*

```
Scanner scanner = new Scanner(System.in);
String userInput = scanner.nextLine();
```

`nextLine()` grabs everything typed into the console until the user enters a carriage return (hits the "Enter" key)

  - Tokens are the size of a line input and labeled `String`

More `Scanner` Methods

To read in a:	Use `Scanner` method
`boolean`	`boolean nextBoolean()`
`double`	`double nextDouble()`
`float`	`float nextFloat()`
`int`	`int nextInt()`
`long`	`long nextLong()`
`short`	`short nextShort()`
`String (appearing in the next line, up to '\n')`	`String nextLine()`
`String (appearing in the line up to next ' ', '\t', '\n')`	`String next()`

`Scanner` Exceptions

- `InputMismatchException`
  - Thrown by all `next`*`Type`*`()` methods
  - Token cannot be translated into a valid value of specified type
  - `Scanner` does not advance to the next token, so that this token can still be retrieved
- Handling this exception
  - Prevent it
    - Test next token by using a `hasNext`*`Type`*`()` method
    - Doesn't advance the token, just checks and verifies type of next token
      - `boolean hasNextBoolean()`
      - `boolean hasNextDouble()`
      - `boolean hasNextFloat()`
      - `boolean hasNextInt()`
      - `boolean hasNextLong()`
      - `boolean hasNextShort()`
      - `boolean hasNextLine()`
      - See the javadocs for more info about `Scanner` methods!
  - Catch it
    - Handle the exception once you catch it

Object Streams

- **`ObjectOutputStream`** class can save entire objects to disk
- **`ObjectInputStream`** class can read objects back in from disk
- Objects are saved in binary format; hence, you use streams
- The object output stream saves all instance variables
  - Example: Writing a **`BankAccount`** object to a file

```
BankAccount b = . . .;
ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("bank.dat"));
out.writeObject(b);
```

Example: Reading a `BankAccount` Object From a File

- `readObject` returns an `Object` reference
  - Need to remember the types of the objects that you saved and use a cast

```
ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

- `readObject` method can throw a `ClassNotFoundException`
  - It is a checked exception
  - You must catch or declare it

Write and Read an `ArrayList` to a File

- Write

```
ArrayList<BankAccount> a = new ArrayList<BankAccount>();
// Now add many BankAccount objects into a
out.writeObject(a);
```

- Read

```
ArrayList<BankAccount> a = (ArrayList<BankAccount>)
      in.readObject();
```

Serializable

- Objects that are written to an object stream must belong to a class that implements the `Serializable` interface.

```
class BankAccount implements Serializable {
    . . .
}
```

- `Serializable` interface has no methods.
- Serialization: process of saving objects to a stream
  - Each object is assigned a serial number on the stream
  - If the same object is saved twice, only serial number is written out the second time
  - When reading, duplicate serial numbers are restored as references to the same object
- Demo: serial

New I/O APIs for the Java Platform

- Buffer: a linear, finite sequence of elements of a specific *primitive* type
  - Consolidate I/O operations
  - Four properties (all having never negative values)
    - Capacity: number of elements it contains (never changes)
    - Limit: index of the first element that should not be read or written
    - Position: index of the next element to be read or written
    - Mark: index to which its position will be reset when the `reset()` method is invoked
  - Invariant: 0 <= *mark* <= *position* <= *limit* <= *capacity*

New I/O APIs for the Java Platform

- Buffers (cont'd)
  - `put` and `get` operations
    - Relative (starting at current position) or absolute
  - `clear`
    - makes a buffer ready for a new sequence of *channel-read* or relative *put* operations: sets limit = capacity, position = 0.
  - `flip`
    - makes a buffer ready for a new sequence of channel-write or relative *get* operations: sets limit = position, then sets position = 0.
  - `rewind`
    - makes a buffer ready for re-reading the data that it already contains: leaves limit unchanged, sets position = 0.
  - `reset`
    - Resets this buffer's position to the previously-marked position

New I/O APIs for the Java Platform

- Channels
  - Connection to an I/O device
    - Have peer **java.io** objects, one of: `FileInputStream`, `FileOutputStream`, `RandomAccessFile`, `Socket`, `ServerSocket` or `DatagramSocket`.
    - Interacts efficiently with buffers
  - `ReadableByteChannel` interface
    - Method `read` reads a sequence of bytes from this channel into the given buffer
  - `WriteableByteChannel` interface
    - Method `write` writes a sequence of bytes to this channel from the given buffer
  - Scattering reads and gather writes
    - operate with sequences of buffers
  - Class `FileChannel`

New I/O APIs for the Java Platform

- File Locks
  - Restricts access to a portion of a file
    - `FileChannel`, position, size
    - Exclusive or shared
- Charsets
  - Package `java.nio.charset`
    - Class `Charset`
      - Methods `decode, encode`
    - Class `CharsetDecoder, CharsetEncoder`
- Demo: fileChannel

Reading

- Eckel chapter 19
- Barnes appendices F & G
- Deitel chapter 17, appendix F

Summary

- Software testing
  - functional testing, design, plan and test cases
  - test harnesses
  - supplying test input
  - evaluation of test results
  - test coverage
  - unit testing - JUnit
- Program execution trace
  - Logging
- Debugging
  - using a debugger
- Java I/O introduction
  - I/O structure
  - classes for reading/writing
  - sequential/random access
  - exceptions
  - general procedure for I/O
- The class Scanner
- Object Streams
  - Serializable
- New Java I/O
  - Buffers, Channels