Java

Lecture 18_Java
Lecture Outline

• Java: COOL on steroids
  – COOL plus many more features
    • Both are strongly typed, object-oriented, garbage collected languages
    • Both designed in the 1990s, so they share a common culture: the ideas that were current at that time
  – We will focus in this slide set on Java history, as well as several features that were too time consuming to include in detail in lectures or projects. These are...
Lecture Outline

• Java History
• Arrays
• Exceptions
• Interfaces
• Coercions
• Threads
• Dynamic Loading & Initialization
• Summary
Java History

• Began as Oak at SUN
  - Originally targeted at set-top devices
    • Was going to make the TV much more interactive (box running Java code)
    • May seem ridiculous now, but only because current TVs are computers in and of themselves
  - Initial development took several years ('91 - '94)
    • But set-to-box market just never took off, so it became clear there was limited potential for Oak
    • But then the Internet happened, and it became clear that there was a need for programming languages that addressed issues specific to the Internet
      - Security: need to be able to share code over the network and not worry that it would crash your device
        » After all, who wrote it?
Java History

- So, retargeted as the Internet language ('94-'95)
  - Beat out other competitors: primarily TCL, Python (ActiveX came later)
    - Ultimately, backing of Sun Microsystems helped it gain a strong presence on the Internet
- The point: every new language needs a “killer app”
  - I.e., every programming language “rides into the world on the back of some new application” that is not served well by existing programming languages
  - Provides incentive for people to learn the new language
  - Recall the earlier lecture on the economy of new programming languages
The People

• James Gosling
  - Principal designer
  - CMU Ph.D.

• Bill Joy
  - ABD from Berkeley (Unix)

• Guy Steele
  - MIT Ph.D.
  - Famous programming language researcher
Influences

• As we mentioned earlier in the semester, languages come into existence in a specific context
  - What is popular, etc

• So it’s normal for new programming languages to borrow heavily from their predecessors
  - New languages are mostly recombinations of ideas from previous languages, but in a new design, with perhaps some innovations thrown in
Influences on Java

• Modula-3
  - type system and commitment to types
  - Modula-3 was an attempt to build a language that would scale to large systems but that was also strongly typed

• Eiffel, Objective C, C++
  - Object orientation, interfaces

• Lisp
  - Java’s dynamic flavor (lots of features)
    • E.g., reflection
Java Design

• From our perspective, COOL plus
  - Exceptions
  - Interfaces
  - Threads
  - Dynamic Loading
  - Other less important ones...

• Java is a BIG language
  - Lots of features
  - Lots of feature interactions
  - It is NOT simple: manual runs to hundreds of pages

One very hard part in language design is getting features, and in particular interactions between them, right!
Arrays

- Assume $B < A$. What happens in the following?

```java
B[] b = new B[10];
A[] a = b;
a[0] = new A();
b[0].aMethodNotDeclaredInA();
```
• Assume $B < A$. What happens in the following?

```java
B[] b = new B[10];
A[] a = b;
a[0] = new A();
b[0].aMethodNotDeclaredInA();
```
• Assume $B < A$. What happens in the following?

```java
B[] b = new B[10];
A[] a = b;
a[0] = new A();
b[0].aMethodNotDeclaredInA();
```

Should be able to call any $b$ method on $b[0]$, but can't, since the object stored there is an $A$
Subtyping Rules in Java

- $B < A$ if $B$ inherits from $A$ as in Cool
- $C < A$ if $C < B$ and $B < A$ as in Cool
- $B[] < A[]$ if $B < A$ not as in Cool

Cool doesn’t even have arrays, so of course this last is not as in Cool

This last rule is unsound!
Subtyping Rules in Java

B < A  \hspace{1cm} \text{if } B \text{ inherits from } A \hspace{1cm} \text{as in Cool}

C < A  \hspace{1cm} \text{if } C < B \text{ and } B < A \hspace{1cm} \text{as in Cool}

B[] < A[]  \hspace{1cm} \text{if } B < A \hspace{1cm} \text{not as in Cool}

This last rule is unsound!

BUT, this is also not the way this is done in other languages that have arrays, objects, and subtyping
What's Going On?

```
B[] b = new B[10];
A[] a = b;
a[0] = new A();
b[0].aMethodNotDeclaredInA();
```

Having multiple aliases (with different types) to updateable locations (e.g., an array), each able to read and write to this memory, is unsound!
Not Unique to Java

• This problem has come up in many different programming languages
  - It’s a fairly subtle issue of type systems
  - And many languages have created a similar problem for their static typing systems by wanting to have subtyping work through arrays
The Standard Solution

- The solution most widely accepted in the programming languages research community
  - Need a different subtyping rule for arrays
- Disallow subtyping through arrays
- Standard solution: Use the rules below
  - The last rule prevents having two aliases of different types from having a reference to the same location, eliminating the problem

\[
\begin{align*}
B < A & \quad \text{if } B \text{ inherits from } A \\
C < A & \quad \text{if } C < B \text{ and } B < A \\
B[] < A[] & \quad \text{only if } B = A
\end{align*}
\]
The Java Solution

• Java fixes the problem differently: by checking each array assignment at runtime for type correctness
  - Whenever an assignment is done into an array, the check: Is the type of the object being assigned compatible with the type of the array?
    • Note this prevents our example: object of type A cannot be assigned to an array with base type B, unless A < B

• Cost: huge overhead on array computations!
  - Every assignment to an array with object base type has a type check on it at runtime!
The Java Solution

• Java fixes the problem differently: by checking each array assignment at runtime for type correctness
  - Whenever an assignment is done into an array, the check: Is the type of the object being assigned compatible with the type of the array?
    • Note this prevents our example: object of type A cannot be assigned to an array with base type B, unless A < B

• But note: arrays of primitive types unaffected
  - Primitive types are not classes
  - And it turns out that many assignments into arrays are done on arrays with primitive base types
A Common Problem

• Deep in a section of code, you encounter an unexpected error, something that could violate an important property of your program
  - Out of memory
  - A data structure that is supposed to satisfy some invariant, but doesn’t
    • E.g., a list that is supposed to be sorted, but is not
  - etc.

• What do you do?
  - I.e., How do you write code that will handle the error gracefully, but without making your program very ugly?
A Popular Solution: Exceptions

- Add a new type (class) of exceptions and
- Add new control constructs for dealing with these:

```plaintext
throw exception
```

causes an exception to be created at the point of the error and propagated out of the program

Causes program execution to halt at that point, and any containing constructs to also throw the exception

So the exception propagates up out of code until it hits a try-catch
A Popular Solution: Exceptions

try{ something } catch(x) { cleanup }

try something (an expression) and if this expression throws an exception, then we catch that exception (there is a binding of \texttt{x} to the exception, somewhat like a \texttt{let} statement) and execute the cleanup code which is expected to handle the exception in some way.

The basic idea behind this design for handling exceptions is that the place where you have the exception might be deep inside the code, and not a very good place for actually dealing with the exception. So get to some place where it’s better to handle the exception, and move on from there.
Example

class Foo {
    public static void main(String[] arg) {
        try { X(); } catch (Exception e) {
            System.out.println("Error!");
        }
    }

    public void X() throws MyException{
        throw new MyException();
    }
}

Note MyException is a class just like any other, except with the special property that it can be “thrown”
Semantics (pseudo-Java)

$T(v) = \text{an exception has been thrown with value } v$
$v = \text{an ordinary object, which behaves like an ordinary object, except when it's thrown, at which point it is treated differently}$

\[
E \vdash e_1 : v
\]

\[
E \vdash \text{try } \{ e_1 \} \text{ catch}(x) \{ e_2 \} : v_2
\]

\[
E \vdash e_1 : T(v_1)
\]

\[
E[x \leftarrow v_1] \vdash e_2 : v_2
\]

\[
E \vdash \text{try } \{ e_1 \} \text{ catch}(x) \{ e_2 \} : v_2
\]
Semantics (pseudo-Java)

\[ T(v) = \text{an exception has been thrown with value } v \]

\[ v = \text{an ordinary object} \]

So this notation distinguishes between when it is normal, “v”, and when it has been thrown, “T(v)”

\[
\begin{align*}
\text{E} & \vdash e_1 : v_1 \\
\text{E} & \vdash \text{try} \{ e_1 \} \text{ catch}(x) \{ e_2 \} : v_1
\end{align*}
\]

\[
\begin{align*}
\text{E} & \vdash e_1 : T(v_1) \\
\text{E}[x \leftarrow v_1] & \vdash e_2 : v_2 \\
\text{E} & \vdash \text{try} \{ e_1 \} \text{ catch}(x) \{ e_2 \} : v_2
\end{align*}
\]
Semantics (pseudo-Java)

\[ T(v) = \text{an exception has been thrown with value } v \]
\[ v = \text{an ordinary object} \]

So this notation distinguishes between when it is normal, “v”, and when it has been thrown, “T(v)”

\[
\frac{E \vdash e_1 : v_1}{E \vdash \text{try } \{ e_1 \} \text{ catch(x)} \{ e_2 \} : v_1}
\]

If an expression has a value and it has not been thrown, then the try-catch block has that same value, since no problems in the try block

\[
\frac{E \vdash e_1 : T(v_1)}{E[x \leftarrow v_1] \vdash e_2 : v_2}
\]

\[
\frac{E \vdash \text{try } \{ e_1 \} \text{ catch(x)} \{ e_2 \} : v_2}{E \vdash e_1 : T(v_1) \quad E[x \leftarrow v_1] \vdash e_2 : v_2}
\]
Semantics (pseudo-Java)

\[ T(v) = \text{an exception has been thrown with value } v \]
\[ v = \text{an ordinary object} \]
So this notation distinguishes between when it is normal, “v”, and when it has been thrown, “T(v)”

\[
E \vdash e_1 : v_1 \\
E \vdash \text{try } \{ e_1 \} \text{ catch(x) } \{ e_2 \} : v_1
\]

If in evaluating \( e_1 \) an exception \( v_1 \) is thrown, then \( v_1 \) is bound to \( x \) in the environment, and the value of the try-catch block is the value of the catch expression

\[
E \vdash e_1 : T(v_1) \\
E[x \leftarrow v_1] \vdash e_2 : v_2 \\
E \vdash \text{try } \{ e_1 \} \text{ catch(x) } \{ e_2 \} : v_2
\]
throw takes an expression e with value v, then it marks that value as being a thrown exception
So how does the rest of the language deal with thrown exceptions?
Like this, which is one example of several rules that are needed: this one says if we are evaluating $e_1 + e_2$ and in evaluating $e_1$ an exception $v_1$ is thrown, then the value of $e_1 + e_2$ is that same exception. Put another way, the exception propagates.
Semantics (Cont.)

\[
\begin{align*}
E & \vdash e : v \\
\hline
E & \vdash \text{throw } e : T(v) \\
\hline
E & \vdash e_1 : T(v_1) \\
\hline
E & \vdash e_1 + e_2 : T(v_1)
\end{align*}
\]

Note that in this case \(e_2\) is not even evaluated. That is, execution stops after evaluation of \(e_1\). There is not evaluation of \(e_2\) on the top.

All forms except catch propagate thrown exceptions
Semantics (Cont.)

\[
\frac{E \models e : v}{E \models \text{throw } e : T(v)}
\]

\[
\frac{E \models e_1 : T(v_1)}{E \models e_1 + e_2 : T(v_1)}
\]

Similarly for all the other forms of expressions: if the evaluation of a subexpression throws an exception, then execution stops, and the value of the expression is that thrown exception.
The only thing that stops the exception from becoming the value of the whole program is if the exception is caught in a try-catch block.
Simple Implementation

- When we encounter a **try**
  - Mark current location in the stack
- When we **throw** an exception
  - Unwind the stack to the first **try**
  - Execute corresponding **catch**
  - This has the unfortunate consequence of creating a cost for a try block even if no exception is thrown
  - extra marking of the stack
- More complex techniques reduce the cost of **try** and **throw**
  - Partly by trading of the cost of **trys** and **throws**
    - Make **try** less expensive since exceptions relatively rare, at cost of more expensive **throw**
Trivia Question

What happens to an uncaught exception thrown during object finalization?
First: What is finalization

- When an object is garbage collected, it is possible to run a method (the finalization method) on that object in order to clean it up before it is actually deallocated
  - The garbage collector invokes finalization methods

- Why would you want to do this?
  - An example: An object has a pointer to a file, but the file was not closed in the code. If the memory is just deallocated, the pointer remains, a potential problem down the road (system has limited number of file handles)
  - Also, leaving an open file hanging around can cause other kinds of problems
First: What is finalization

- When an object is garbage collected, it is possible to run a method (the `finalization method`) on that object in order to clean it up before it is actually deallocated
  - The garbage collector invokes finalization methods

- Why would you want to do this?
  - So the finalization method in this case would properly close the file, deallocate the file handle, then deallocate the memory for the object itself
  - In general these things often clean up any resources that the object has
So, back to the Trivia Question

What happens to an uncaught exception thrown during object finalization?

An exception thrown during finalization is thrown by code invoked by the garbage collector, so it is unpredictable when it will run. It’s not clear exactly where that exception should be handled.
So, back to the Trivia Question

What happens to an uncaught exception thrown during object finalization?

Answer: Nothing. No one catches it. It is simply dropped. Any exceptions thrown during finalization are simply thrown away unless the finalization method handles them itself.
Type Checking

• One of the nice innovations in Java is that they are actually part of the method interface and thus they are checked by the compiler

• Methods must declare types of exceptions they may raise

  public void X() throws MyException

  - Checked at compile time

    • Why? Designers observed that there are many different exceptions that can be raised by Java programs, and people easily lost track of what possible exceptions could be raised (as well as what exceptions they had to handle)
Type Checking

• Methods must declare types of exceptions they may raise

   public void X() throws MyExcption

   - Checked at compile time

   • Moreover, when they added the requirement that programmers had to declare every exception they could raise, they discovered many places in the compiler code itself where exceptions were raised but not properly handled

   • So this lead to better error handling in the Java compiler itself

   • And most people agree that this is a good feature because it helps people write more robust code
Type Checking

• Methods must declare types of exceptions they may raise

  public void X() throws MyException

  - There are some exceptions to this rule: some exceptions need not be part of the method signature
    • There are some kinds of run time errors that don’t have to be part of the method signature because it’s very hard to check statically that the method would never raise them
    • e.g., dereferencing null, integer overflow

• Other mundane type rules
  - throw must be applied to an object of type Exception
Type Checking

• Methods must declare types of exceptions they may raise

  public void X() throws MyException

  - Checked at compile time
  - Some exceptions need not be part of the method signature
    • e.g., dereferencing null

• Other mundane type rules
  - throw must be applied to an object of type Exception
Java Exceptions: Overall

- Exception handling is done very well in Java
- And the idea of requiring declaring in method signatures the type of exceptions potentially being raised was a new one introduced by Java
Interfaces in Java

- Specify relationships between classes without inheritance

```java
interface PointInterface {
    void move(int dx, int dy);
}

class Point implements PointInterface {
    void move(int dx, int dy) {
        ... }
}
```
Interfaces

“Java programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to Object”

In other words interfaces play the same role as multiple inheritance in C++, because classes can implement multiple interfaces

```java
class X implements A, B, C { ... }
```

Note that though X does not inherit anything from A, B, or C, X can be treated as either an A object, or a B object, or a C object! So it’s almost as if X had three superclasses
Why is this Useful?

• A graduate student may be both a University employee and a student

```java
class GraduateStudent implements Employee, Student{
    ...
}
```

• No good way to incorporate Employee, Student methods for grad students with single inheritance
  - This is not ideal: you don’t get code reuse as you do with inheritance.
Implementing Interfaces

• Biggest implementation difference between interfaces and inheritance is that methods in classes implementing interfaces need not be at fixed offsets
  - Though I use the term “need not” above, the reality is that in many cases it’s simply not possible to have methods be at fixed offset
  - For this reason it’s not as efficient to implement interfaces as it is to implement inheritance
  - This is one of the reasons that languages like Java have both
Implementing Interfaces: Example

```java
interface PointInterface {
    void move(int dx, int dy);
}

class Point implements PointInterface {
    void move(int dx, int dy) {
        ... }
}

class Point2 implements PointInterface {
    void dummy() {
        ... }
    void move(int dx, int dy) {
        ... }
}
```

Where do we put the `move` method? Clearly not in the order in which they are declared, otherwise they are not at the same offset in both classes.
interface PointInterface {void move(int dx, int dy); }  

class Point implements PointInterface {
    void move(int dx, int dy) { ... }
}
class Point2 implements PointInterface, Drawable {
    void dummy() { ... }
    void move(int dx, int dy) { ... }
}

One might imagine fixing this by having the compiler do multiple passes and arrange for all implementing classes to have, say, the move method at the same offset. But this doesn’t work when you have classes implementing multiple interfaces.
Implementing Interfaces: Example

interface PointInterface {void move(int dx, int dy);}

class Point implements PointInterface {
    void move(int dx, int dy) { ... }
}
class Point2 implements PointInterface, Drawable {
    void dummy() { ... }
    void move(int dx, int dy) { ... }
}

In general, there is no total ordering we can give all the methods in all the interfaces so that they can be maintained across all the classes that implement those interfaces (at least not without knowing beforehand all of the classes that are declared and all the interfaces that are declared -- which prevents extending interfaces)
Implementing Interfaces (cont.)

• Since methods not at fixed interfaces, how can we implement interfaces?

• Dispatches `e.f( ... )` where `e` has an interface type are more complex than usual

• One (quite inefficient) approach:
  - Each class implementing an interface has a lookup table `method names -> methods`
  - Hash method names for faster lookup
    • hashes computed at compile time
Implementing Interfaces (cont.)

• Since methods not at fixed interfaces, how can we implement interfaces?

So object has pointer to “normal” method table, which itself has a pointer to the “interface method lookup” table.
Coercions

- Coercions occur in many languages
- We focus (as expected) on how this is done in Java
Coercions

- Java allows primitive types to be *coerced* in certain contexts
  - I.e., converted from one type to another
- Ex. In `1 + 2.0`, the `int 1` is widened to a `float 1.0`
  - No way to do the addition without converting one of the two to the type of the other (think hardware, binary representation)
- Think of coercion as really just a primitive function the compiler inserts for you (e.g., `int2float()`)  
  - Most languages have extensive coercions between base numeric types
Coercions

• Java allows primitive types to be coerced in certain contexts
  - I.e., converted from one type to another
• Ex. In 1 + 2.0, the int 1 is widened to a float 1.0
  - No way to do the addition without converting one of the two to the type of the other (think hardware, binary representation)
• Think of coercion as really just a primitive function the compiler inserts for you (e.g., int2float())
  - Really just a convenience for the programmer
Coercions & Casts

• Java distinguishes two kinds of coercions & casts:
  - *Widening:* always succeed (*int* → *float*)
  - *Narrowing:* may fail if data can’t be converted to desired type (*float* → *int*, downcasts)
    • For narrowing casts where there isn’t a clear mapping (e.g., trying to convert 2.5 to *int*, do we truncate? round up?), Java will complain and won’t do it
Coercions & Casts

• Java distinguishes two kinds of coercions & casts:
  - **Downcast** example:
    • \( B < A \) and
    • \( x \) has type \( A \) and
    • programmer tries to cast \( x \) to type \( B \): \((B) \ x\)
    • This will compile, but fail at run-time (and throw an exception) if object \( x \) holds at time of cast is not of type \( B \)
Coercions & Casts: The Rule in Java:

- Narrowing casts must be explicit
  - You must put the type cast in the code so that it’s obvious that you really want to do it (and even then it might not allow you to)

- Widening casts/coercions can be implicit
Another Trivia Question

What is the only type in Java for which there are no explicit or implicit coercions or casts defined?
Another Trivia Question

What is the only type in Java for which there are no explicit or implicit coercions or casts defined?

Solution: boolean
Coercions

• Coercions can lead to surprising behavior
• And it’s questionable whether they are a good thing
  - Certainly a programmer convenience
  - And definitely ubiquitous
  - But can lead to behavior that is not what the programmer expected
    • Which seems to argue for this being more harm than benefit
Example: Coercions in PL/I

- We’ve talked about PL/I in this class earlier
  - Lots of features
  - PL/I had very extensive casts and coercions
- Let $A$, $B$, $C$, be strings of 3 characters
  - And note that the length 3 is part of the type

  $B = '123'$

  $C = '456'$

  $A = B + C$

- What is $A$?
Example: Coercions in PL/I

\[ B = '123' \]
\[ C = '456' \]
\[ A = B + C \]

- What is \( A \)? Well, in PL/I, the + is going to be interpreted as integer addition, so \( B \) will get converted to the integer 123, \( C \) will be converted to the integer 456, and when added we get 579, so result of rhs expression is the integer 579
Example: Coercions in PL/I

B = '123'
C = '456'
A = B + C

• What is \textbf{A}? BUT, the left hand side is a string of 3 characters, so the number 579 has to be cast back to a string of 3 characters.

• Cast happens in two steps:
  - 579 is first cast to a string of the default length, which is 6: 579 -> "579"
Example: Coercions in PL/I

B = ‘123’
C = ‘456’
A = B + C

• What is A? BUT, the left hand side is a string of 3 characters, so the number 579 has to be cast back to a string of 3 characters.

• Cast happens in two steps:
  - Then string of 6 characters is cast to a string of 3 characters by taking the first three characters, leaving the 3 character string with three spaces “   ”
Example: Coercions in PL/I

• We’ve talked about PL/I in this class earlier
  - Lots of features
  - PL/I had very extensive casts and coercions

• Let \( A, B, C \), be strings of 3 characters
  - And note that the length 3 is part of the type

\[
\begin{align*}
B &= '123' \\
C &= '456' \\
A &= B + C
\end{align*}
\]

• What is \( A \)? A string consisting of three spaces
  - Probably not what the programmer intended
Java Threads

- Java has concurrency built in through *threads*
  - Each thread has its own program counter and stack (local variables), but may refer to a common area in the heap
  - Think of multiple instantiations of a program appearing to run simultaneously, but sharing the same OS process
  - Some kind of *thread scheduler* starts and stops the various thread instances (context switching) in (basically) round robin fashion, so in reality only one thread is executing at once
    - It is actually nondeterministic the order in which the thread instances will be executed, and how many program statements will execute during a given time slice
Java Threads

- Java has concurrency built in through *threads*
  - Each thread has its own program counter and stack (local variables), but may refer to a common area in the heap
  - Think of multiple instantiations of a program appearing to run simultaneously, but sharing the same OS process
  - Bottom line: the instructions in the various threads interleave in a completely arbitrary order
Java Threads

• In Java, all Thread objects have class `Thread`
  - More specifically, a class must inherit from `Thread` in order to be a thread
  - `start()` and `stop()` methods (inherited from `Thread`) for beginning and ending the thread

• One thing threads can do: `synchronize` on objects
  - Synchronization obtains a `lock` on the object through the `synchronized` construct:
    
    ```java
    synchronized (x) { e }
    ```
Java Threads

• Synchronization obtains a *lock* on the object through the **synchronized** construct:

  
  ```java
  synchronized (x) { e }
  ```

• In Java, this code means that the program will acquire a lock on `x`, before it executes `e`.
  - So the sequence is:
    - 1) lock `x`,
    - 2) evaluate `e`,
    - 3) unlock `x`

• While the program is evaluating `e`, it will hold the lock on `x`, preventing any other thread from accessing it.
Java Threads

• This mechanism is the only way in Java to get synchronization between multiple threads
  - Allowing one to control the set of interleavings of the thread statements
    • Basically, holding the lock means that no other thread can access object \( x \) until the lock is released
    • So note that two threads could be executing the same code section on two different objects (presumably each holding a lock on the object on which they are working)

• One shorthand in Java: synchronization can be put on methods
  - E.g., \texttt{synchronized f(...)} \{   \}
  - which means that when \( f \) is called, the \texttt{this} object is (implicitly) locked
class Simple {
    int a = 1, b = 2;
    void to() { a = 3; b = 4;}
    void fro() { println("a = " + a + ", b = " + b); }
}

Suppose two threads of class Simple, one which calls \texttt{to()} and one which calls \texttt{fro()}. What is printed?

Assume thread 1 calls \texttt{to()} and thread 2 calls \texttt{fro()}
Example (from the Java Spec)

class Simple {
    int a = 1, b = 2;
    void to() { a = 3; b = 4;}
    void fro() { println("a = " + a + ",b = " + b); }
}

If thread 1 to() runs to completion, then thread 2 fro() runs, the output is “a = 3, b = 4”
Example (from the Java Spec)

class Simple {
    int a = 1, b = 2;
    void to() { a = 3; b = 4;}
    void fro() { println("a = " + a + ", b = " + b); }
}

If thread 2 fro() runs to completion, then thread 1 to() runs, the output is "a = 1, b = 2", after which to() sets a = 3 and b = 4
Example (from the Java Spec)

class Simple {
    int a = 1, b = 2;
    void to() { a = 3; b = 4; }
    void fro() { println("a = " + a + ", b = " + b); }
}

A somewhat stranger possibility: the threads interleave in a nontrivial way:

to() paused before assigning 4 to b

to()  a = 3

fro()  time  "a = 3, b = 2"

a partial update: a updated but b not
Example (Cont.)

class Simple {
    int a = 1, b = 2;
    void synchronized to() { a = 3; b = 4;}
    void fro() { println("a = " + a + ", b = " + b); }
}

Note this use of synchronized is not correct, and is actually an example of a synchronization error that even seasoned professional Java programmers often make.
Example (Cont.)

class Simple {
    int a = 1, b = 2;
    void synchronized to() { a = 3; b = 4; }
    void fro() { println("a = " + a + ", b = " + b); }
}

So, this example is actually fairly instructive.

The assumption: in the heap there is only a single object, s, of class Simple
Example (Cont.)

```java
class Simple {
    int a = 1, b = 2;
    void synchronized to() { a = 3; b = 4; }
    void fro() { println("a = " + a + ", b = " + b); }
}
```

Assume `to()` is paused at this point, and `fro()` starts to run. Even though `to()` has a lock on `s`, `fro()` can access it! Why? Because `fro()` doesn't check the lock - it's not synchronized!
class Simple {
    int a = 1, b = 2;
    void synchronized to() { a = 3; b = 4; }
    void fro() { println("a = "+a+", b = "+b); }
}

lock s
heap
a = 3
b = 4
unlock s

So this particular attempted fix has achieved nothing

"a = 3, b = 2"
class Simple {
    int a = 1, b = 2;
    void synchronized to() { a = 3; b = 4; }
    void fro() { println("a = " + a + ", b = " + b); }
}

heap

s

lock s
a = 3
b = 4
unlock s

to()  fro()
"a = 3, b = 2"

All possible interleavings of the two methods still exist if only one of the two is synchronized
Why is this error common?

• Well, because frequently people think that reads are OK
  - reads change nothing, so they reason that one can always read from things in parallel
  - It’s the writes they think they need to worry about, so they synchronize the methods that write

• But as we see in this example, having only one half of the accesses (in our example, writes) synchronized doesn’t help
  - Because synchronization only works if all accessing methods are checking the lock
Two threads call `to()` and `fro()`. What is printed? In this case only two possibilities: “a = 1, b = 2” or “a = 3, b = 4”
Semantics

• Even without synchronization, a variable should only hold values written by some thread
  - Java guarantees that writes of values are atomic
    • E.g., if thread 1 writes \( a = 3.14 \) and thread 2 writes \( a = 2.78 \), the value of \( a \) should be either 3.14 or 2.78, not some combination of the two, which could happen if parts of the bits are written by one thread and parts written by the other thread before the first is finished writing the value
  - Rule is violated for doubles, though
    • Why? Because doubles require two words (hence the term “double”), which means that writing the value of a double requires two instructions. So our example shows that one thread might write the high word of the double and another thread the low word
Semantics

- Even without synchronization, a variable should only hold values written by some thread
  - Java guarantees that writes of values are atomic
    - E.g., if thread 1 writes \( a = 3.14 \) and thread 2 writes \( a = 2.78 \), the value of \( a \) should be either 3.14 or 2.78, not some combination of the two, which could happen if parts of the bits are written by one thread and parts written by the other thread before the first is finished writing the value.
  - Rule is violated for doubles, though
    - Such a value is called an *out of thin air* value (I kid you not)
    - These are obviously bad, but Java allows this as a concession to hardware and performance. Problem prevented by declaring the double to be *volatile*
Semantics

- In general Java concurrency semantics are difficult to understand in detail, particularly as to how they might be implemented on certain machines
  - The out of thin air values are one aspect of this
  - There are several other aspects of it
  - This is not Java’s fault: concurrency semantics are hard - this is at the frontier of research
    - We don’t totally understand what the right thing to do is to specify the behavior of languages in concurrent settings
    - We understand some aspects, but in a language as feature rich as Java, it’s just not clear how some things should be implemented on some machines
Semantics

• In general Java concurrency semantics are difficult to understand in detail, particularly as to how they might be implemented on certain machines
  - The out of thin air values are one aspect of this
  - There are several other aspects of it
  - This is not Java’s fault: concurrency semantics are hard - this is at the frontier of research
    • There has been a good deal of research specifically on this aspect of Java, as it was the first major programming language to have built in concurrency along with all the other modern features that we like
      - Bottom line: if you’re doing basic concurrency in Java, fine. Complex stuff, well, that can be tricky
Dynamic Loading

• Java allows classes to be loaded at run time
  - So you can actually add functionality to a Java program while it's running by loading a new class
    • This creates extra issues with type safety and security, because now there is a distinction between compile time and load time
  - Type checking of source takes place at compile time
Dynamic Loading

- Java allows classes to be loaded at run time
  - But what the loader loads is not source, but bytecode
    - And this bytecode could have come from an untrusted source, perhaps from a compiler that did not do type checking
      - So this bytecode might not satisfy type assumptions of the Java implementation
  - Thus it must be checked: bytecode verification takes place at run time (this is really just type checking of bytecode)
    - Procedure is different because code is much lower level, so algorithms are a bit different, but it’s still just type checking
Dynamic Loading

• Loading policies handled by a ClassLoader
  - A special Java class that determines what can be loaded
    • A problem in early versions of Java, where it was discovered that an attacker could get control of the ClassLoader and install its own ClassLoader that was much more permissive and subvert the system.
    - This issue was fixed long ago

• Another interesting feature: classes may also be unloaded
  - But poorly specified in the definition
    • So it’s not clear what happens when you unload a class, or what happens to the existing object when you do this.
Initialization

• Initialization in Java is complex
  - Shouldn’t be surprising, since initialization in COOL is also fairly complex
  - Java is a superset of COOL, so it has everything in COOL plus much more
  - Main source of complication is concurrency
    • Though other features add to this complication
    • In fact, a good way to understand the features of a new language is to understand how it does object and class initialization
      - Because during initialization, all the features of the language are interacting, so get to learn what all of those interactions are and how they are sorted out in order to have a well defined initialization procedure
Class Initialization

- **Class initialization**: how the object that actually represents a class, gets initialized, when that class is first brought into the program
  - Note that this idea of a class object is NOT something that COOL has
  - A *class object* is the object for a class - it represents the class
  - It is NOT an instance of the class, rather it is an object that IS the class
    - It has all the information about the class
      » What the type if the class is
      » What the fields are, etc.
  - Used for introspection or reflection
  - Necessary in Java for features like dynamic loading
    - So you can query the class for the kinds of fields and methods
Class Initialization

- A **class** is initialized when a symbol in the class is first used, **not** when the class is loaded
  - Reason is that this delays potential initialization errors to a predictable point (i.e., when something in the class is referenced)
  - Makes the error repeatable and predictable
  - If instead error happened when the class was loaded, well this can happen at lots of different times, making the error nondeterministic
Class Initialization Procedure (Partial)

When you load a class, the first thing you have to do is initialize the class object:

1. Lock the class object for the class
   • Wait on the lock if another thread has locked it

2. If the same thread is already initializing this class, release lock and return
   • How could this happen: recursive class (and other ways)

3. If class already initialized, return normally

4. Otherwise, mark initialization as in progress by this thread and unlock class
5. Initialize superclass, fields (in textual order)
   • But initialize static, final fields first
   • Give every field a default value before initialization
   • Step 5 essentially the same as in COOL

6. Any errors result in an incorrectly initialized class, mark class as erroneous

7. If no errors, lock class, label class as initialized, notify threads waiting on class object, unlock class

We've skipped a few things here, but this is the general idea
Features and Feature Interactions

• General point: In any system with N features, there are potentially $N^2$ pairwise feature interactions
  - So interactions grow superlinearly in the number of features added

• Big, featureful systems are hard to understand!
  - Including programming languages
Summary

• Java is pretty well done
  - By production language standards, very well done

• Java brings many important ideas into the mainstream
  - Strong static typing
  - Garbage collection
  - These ideas had been around, but not used in a large production language

• But Java also
  - Includes many features we don’t understand
  - Has a lot of features