Announcements

• A5 due date postponed!
• New due date Monday Dec 8, 11:59pm

CS2110 F08 Quiz 3
Answer briefly the following questions.
1. Explain the difference between shadowing and overriding.

   Shadowing occurs when a field of a class has the same name and type as a field of a superclass, or when a local variable of a class has the same name and type as a field of that class. It is useful only in limited circumstances (e.g. initializing a field with a parameter). Overriding occurs when a method of a class has the same name and signature as a method of a superclass, and is one of the most useful features of object oriented languages.

2. Explain the difference between interfaces and abstract classes.

   Interfaces may contain only constants and abstract methods (method signatures but no implementations). Abstract classes may contain fields and both abstract and concrete methods. Neither interfaces nor abstract classes can be directly instantiated with new.

3. Explain the difference between static and dynamic types.

   Expressions have static types. The type of any expression is known to the Java compiler at compile time, before the program is run. Objects have dynamic types. Dynamic types are only known at runtime. An object receives its dynamic type when it is created with new.

Today

• Class file format
• Class loading and initialization
• Object initialization
• Method dispatch
• Exception handling
• Java security model
  – Bytecode verification
  – Stack inspection
Instance Method Dispatch

\texttt{x\_foo(...)}

• compiles to \texttt{invokevirtual}
• Every loaded class knows its superclass
  – name of superclass is in the constant pool
  – like a parent pointer in the class hierarchy
• bytecode evaluates arguments of \texttt{x\_foo(...)}, pushes them on the stack
• Object \texttt{x} is always the first argument

Instance Method Dispatch

\texttt{invokevirtual foo (\ldots)V}

• Name and type of \texttt{foo(...)} are arguments to \texttt{invokevirtual} (indices into constant pool)
• JVM retrieves them from constant pool
• Gets the dynamic (runtime) type of \texttt{x}
• Follows parent pointers until finds \texttt{foo(...)V} in one of those classes – gets bytecode from code attribute

Instance Method Dispatch

• Creates a new \textit{stack frame} on runtime stack around arguments already there
• Allocates space in stack frame for locals and operand stack
• Prepares locals (int=0, ref=null), empty stack
• Starts executing bytecode of the method
• When returns, pops stack frame, resumes in calling method after the \texttt{invokevirtual} instruction

Stack Frame of a Method

```
local variable array

\begin{tabular}{c c c c}
\hline
String & Hash-table & P1 & Object \\
\hline
\end{tabular}

parameters

other locals

\hline
maxLocals

\hline
maxStack

\hline
```

Exception Handling

• Each method has an \textit{exception handler table} (possibly empty)
• Compiled from \texttt{try/catch/finally}
• An exception handler is just a designated block of code
• When an exception is thrown, JVM searches the exception table for an appropriate handler that is in effect
• \texttt{finally} clause is executed last
Exception Handling

- Finds an exception handler → empties stack, pushes exception object, executes handler
- No handler → pops runtime stack, returns exceptionally to calling routine
- finally clause is always executed, no matter what

Example

```java
Integer x = null;
Object y = new Object();
try {
    x = (Integer)y;
    System.out.println(x.intValue());
} catch (ClassCastException e) {
    System.out.println("y was not an Integer");
} finally {
    System.out.println("finally!");
}
```
Integer x = null; Object y = new Object();
try {
    x = (Integer) y;
    System.out.println(x.intValue());
} catch (ClassCastException e) {
    System.out.println(x.intValue());
    x = (Integer) y;
} finally {
    System.out.println("finally!");
} catch (NullPointerException e) {
    System.out.println(x.intValue());
    x = (Integer) y;
} finally {
    System.out.println("finally!");
}
Java Security Model

- Bytecode verification
  - Type safety
  - Private/protected/package/final annotations
  - Basis for the entire security model
  - Prevents circumvention of higher-level checks
- Secure class loading
  - Guards against substitution of malicious code for standard system classes
- Stack inspection
  - Mediates access to critical resources

Bytecode Verification

- Performed at load time
- Enforces type safety
  - All operations are well-typed (e.g., may not confuse refs and ints)
  - Array bounds
  - Operand stack overflow, underflow
  - Consistent state over all dataflow paths
- Private/protected/package/final annotations

Types in the JVM

Typing of Java Bytecode

A form of dataflow analysis or abstract interpretation performed at load time
- Annotate the program with information about the execution state at each point
- Guarantees that values are used correctly
Example

Preconditions for safe execution:
- local 3 is an integer
- stack is not full

Effect:
- push integer in local 3 on stack
Problem: mobile code is not trustworthy!
- We often have trusted and untrusted code running together in the same virtual machine
  - e.g., applets downloaded off the net and running in our browser
- Do not want untrusted code to perform critical operations (file I/O, net I/O, class loading, security management,...)
- How do we prevent this?

Early approach: signed applets
- Not so great
  - everything is either trusted or untrusted, nothing in between
  - a signature can only verify an already existing relationship of trust, it cannot create trust
- Would like to allow untrusted code to interact with trusted code
  - just monitor its activity somehow
Mobile Code

Q) Why not just let trusted (system) code do anything it wants, even in the presence of untrusted code?

A) Because untrusted code calls system code to do stuff (file I/O, etc.) – system code could be operating on behalf of untrusted code.

Runtime Stack

Maybe we want to disallow it
- the malicious applet may be trying to erase our disk
- it's calling system code to do that

Or, maybe we want to allow it
- it may just want to write a cookie
- it called System.cookieWriter
- System.cookieWriter knows it's ok

Maybe we want to allow it for another reason
- all running methods are trusted

Q) How do we tell the difference between these scenarios?
A) Stack inspection!
• An invocation of a trusted method, when calling another method, may either:
  – *permit* \( R \) on the stack above it
  – *forbid* \( R \) on the stack above it
  – *pass* permission from below (be transparent)

• An instantiation of an untrusted method must *forbid* \( R \) above it

• When about to execute \( R \), look down through the stack until we see either
  – a system method permitting \( R \) -- do it
  – a system method forbidding \( R \) -- don’t do it
  – an untrusted method -- don’t do it

• If we get all the way to the bottom, do it (IE, Sun JDK) or don’t do it (Netscape)

Case A: \( R \) is not executed

Case B: \( R \) is executed

Case C: \( R \) is executed

**Conclusion**

Java and the Java Virtual Machine:
Lots of interesting ideas!