Java Loose Ends
What Else?

• A few Java issues introduced earlier deserve a more in-depth treatment:
  – Try-Catch and Exceptions
  – Members (static vs. instance)
  – Nested interfaces and classes
  – Access modifiers
  – “Final”
Exceptions

• An exception indicates a problem with an application that entails (in Java) a dramatic change of control flow

• Vocabulary: Exceptions (and Errors) are
  – “thrown” by a component implementation
  – “caught” by a client
Syntax of Try-Catch

• In a client, a **try-catch** statement is used to catch exceptions:

```java
try {
    statements
} catch(exceptionType1 identifier1) {
    handler for type1
} catch(exceptionType2 identifier2) {
    handler for type2
}
...
```
Execution of Try-Catch

• If nothing is thrown during execution of the statements in the *try* block:
  – The *try* block finishes successfully
  – All *catch* clauses are ignored (skipped)
  – Execution continues after the *try-catch*
Catching an Exception

• If something is thrown during execution of the statements in the `try` block:
  1. The rest of the code in the `try` block is skipped
  2. The `catch` clauses are examined top to bottom for the first matching `catch`
  3. If an appropriate `catch` clause is found:
     - The body of the `catch` clause is executed
     - The remaining `catch` clauses are skipped
  4. If no such `catch` clause is found:
     - The exception is `thrown` to the outer block, which is either
       - A `try` block (that potentially handles it, in the same manner)
       - A method body (resulting in it being thrown to its client)
Use Exceptions in Your Design?

• **Best practice** suggests exceptions should be reserved for *unexpected* situations:
  – Problems external to the application
  – Resource exhaustion
  – Problems that cannot be handled with checkable preconditions in contracts
Use Exceptions in Your Design?

• **Best practice** suggests exceptions should be reserved for *unexpected* situations:
  – Problems external to the application
  – Resource exhaustion
  – Problems that cannot be handled with checkable preconditions

Example: there is a hardware problem with a disk drive.
Use Exceptions in Your Design?

• **Best practice** suggests exceptions should be reserved for *unexpected* situations:
  – Problems external to the application
  – Resource exhaustion
  – Problems that cannot be handled with checkable preconditions

Example: the JVM is out of memory for this application.
Use Exceptions in Your Design?

• **Best practice** suggests exceptions should be reserved for *unexpected* situations:
  – Problems external to the application
  – Resource exhaustion
  – Problems that cannot be handled with checkable preconditions in contracts

*Example:* a file does not exist—because it has been deleted *after* its existence has already been “confirmed”.
Hierarchy of Classes

- Throwable
  - Error
  - Exception
Hierarchy of Classes

- Throwable
  - Error
    - “Unrecoverable”: hardware, JVM, or application error (e.g., “out of memory”)
  - Exception
    - “Recoverable”: application problem (e.g., “file not found”)
Hierarchy of **Error** Classes

```
Throwable

Error

VirtualMachine-Error
AssertionError

OutOfMemory-Error

Exception
```
Hierarchy of Error Classes

This is thrown by an `assert` statement when the condition is `false`: something is wrong with your program.
Hierarchy of **Error** Classes

There are many more subclasses of **Error**!
Hierarchy of Exception Classes

- Throwable
  - Error
    - VirtualMachineError
    - OutOfMemoryError
  - AssertionError
  - Exception
    - RuntimeException
      - NullPointerException
      - IndexOutOfBoundsException
      - ArithmeticException
    - IOException
Hierarchy of **Exception** Classes

There are many more subclasses of **Exception**, **RuntimeException**, and **IOException**!
Hierarchy of Exception Classes

With `RuntimeError` and its subclasses, something is wrong with your program.
Unchecked vs. Checked

- **Unchecked** exceptions are:
  - `Error` and its subclasses
  - `RuntimeException` and its subclasses
- The rest are **checked** exceptions
  - “Checked” means that the compiler checks that a method whose body contains a statement that might throw the exception either **catches** it, or explicitly “propagates it up the call chain” by declaring that it also **throws** the exception
Unchecked vs. Checked

Throwable

Error
- VirtualMachineError
- OutOfMemoryError
- AssertionError

Exception
- RuntimeException
- IOException
- NullPointerException
- IndexOutOfBoundsException
- ArithmeticException
Members

• A class may have different kinds of members:
  – Variables/fields/data members
  – Constructors
  – Methods
  – Nested classes

• All except constructors may be either static members or instance members
Members

• A class may have different kinds of members:
  – Variables/fields/data members
  – Constructors
  – Methods
  – Nested classes

• All except constructors may be either static members or instance members

Static methods and instance methods have already been discussed...
Members

• A class may have different kinds of members:
  – Variables/fields/data members
  – Constructors
  – Methods
  – Nested classes

• All except constructors may be either static members or instance members

Static members of a class are also called class members, for reasons that will presently become clear.
Static vs. Instance Variables

• At run-time, a Java program has separate representations for:
  – All *static variables* for each *class* \( C \)
  – All *instance variables* for each *instance* of \( C \), i.e., for each object with dynamic type \( C \)

• Bytecode for \( C \) constructors, methods, and nested classes is part of the run-time representation of class \( C \)
Example Class C

```java
class C implements cInterface {
    private static String sVar;
    private int iVar;
    public C(String s, int i) {
        C.sVar = s;
        this.iVar = i;
    }
    public static void sMethod(...) {...}
    public void iMethod(...) {...}
}
```

```java
cInterface x1 = new C("foo", 1);
cInterface x2 = new C("bar", 2);
```
Example Class C

```java
public class C implements cInterface {
    private static String sVar;
    private int iVar;
    public C(String s, int i) {
        C.sVar = s;
        this.iVar = i;
    }
    public static void sMethod(...) {...}
    public void iMethod(...) {...}
}
... 
cInterface x1 = new C("foo", 1);
cInterface x2 = new C("bar", 2);
```

The relevant info for upcoming run-time pictures is the part shown in red.
Run-Time Picture

```
C

instanceof

sVar

"bar"

instanceof

x1

1

iVar

2

 iVar

x2
```
There is a run-time representation of each class that holds all its static variables.
Type erasure: the picture would be the same even if C were generic, say C<T>, and x₁ and x₂ had different T.
Surprise! This means that even if \( C \) were generic, there would still be only one “copy” of \( sVar \).
There is a run-time representation of each instance that holds all its *instance variables*. 
Run-Time Picture

Each instance “knows” its dynamic type at run-time.

(instanceof)

C

sVar

"bar"

instanceof


instanceof

x1

1

iVar

2

iVar

x2
Static Initialization Blocks

• To initialize static variables in a class, you may write a *static initialization block* that looks like this:

```java
static {
    // Code to initialize static variables
}
```

• This code is automatically executed when the class is *loaded*, i.e., *once* at the very beginning of program execution.
Nested Interfaces

• An interface may be *nested* within another interface
  – Example (from OSU CSE components)
    \[ \text{Map.Pair<K,V>} \]
  – Example (from Java libraries)
    \[ \text{Map.Entry<K,V>} \]
Nested Classes

• A class that is *nested* as a member of another class may be:
  – A *static nested class*
  – An *instance nested class*, which is called an *inner class*
Static Nested Class

- A **static** nested class does *not* have access to *instance* members of its enclosing class
  - Effectively, it’s a top-level class declared inside another class since it “logically belongs”
  - Example (from OSU CSE components): `MapSecondary.SimplePair<K,V>`
  - Example (from Java libraries): `AbstractMap.SimpleEntry<K,V>`
Inner Class

• Each *instance* of an inner class belongs to an *instance* of its enclosing class
  – Has access to generic parameters, variables, methods, etc., of its enclosing instance
  – Examples (from OSU CSE components):
    Stack2.Node
    Stack2.Stack2Iterator
Run-Time Picture

Diagram showing a structure with nodes labeled 'top', '2', 'data', 'next', '6', and '18'. Arrows indicate connections between the nodes.
These two instances of Node “belong to” the Stack2 instance that created them.
Access Modifiers

• There are four access modifiers in Java. In decreasing order of visibility (but not presented in this order on the following slides), they are:
  – `public`
  – `protected`
  – default (“package private”)
  – `private`
Access Modifiers

- There are four access modifiers in Java.
  - public
  - protected
  - default ("package private")
  - private

In decreasing order of visibility (but not presented in this order on the following slides), they are:

Only public and “package private” apply to top-level units, i.e., interfaces and classes; interface members can be public (the default) or private (not used in OSU components); class members can be any of the four.
public

• A *top-level unit* declared `public` is accessible from anywhere
  – So long as it is in scope, e.g., via `import`, which henceforth goes without saying

• A *member* declared `public` is accessible from anywhere
  – So long as the top-level unit it’s in is also accessible, which henceforth goes without saying
Default ("Package Private")

- A **top-level unit** declared without any access modifier is accessible from anywhere within the same package
  - The default is also called **package private** accessibility (though probably it should be called “package public”)

- A **class member** declared without any access modifier is accessible from anywhere within the same package
Default (Package Private)

- A **top-level unit** declared without any access modifier is accessible from anywhere within the same package.
  - The default is also known as **package private** accessibility (though probably it should be called “package public”)
- A **class member** declared without any access modifier is accessible from anywhere within the same package.

This offers little protection against unintended access, because anyone can declare their class to be in a given package (e.g., `pkg`) simply by adding this as its first line:

```
package pkg;
```
Default (“Package Private”)

- A *top-level unit* declared without any access modifier is accessible from anywhere within the same package.
  - The default is also called “package private accessibility”.

- A *class member* declared without any access modifier is accessible from anywhere within the same package.

Recall: an *interface member* declared without any access modifier is *public*. 
protected

• A class member declared protected is accessible from within any subclass, and from anywhere in the same package

• Example: the JUnit test fixture pattern in which there is a protected method wrapping each constructor of the UUT (which is meant to be overridden in a subclass specific to that UUT)
**protected**

- A *class member* declared **protected** is accessible from within any subclass, *and* from anywhere in the same package.

- Example: the JUnit test fixture pattern in which there is a **protected** method wrapping each constructor of the UUT (which is meant to be overridden in a subclass specific to that UUT).

So, like the default, this offers little protection against unintended access (despite its optimistic name).
private

• A class (or interface) member declared private is accessible only from within the class (or interface) containing the private member

• Best practice is to make all members in interfaces public and all static and instance variables in classes private, and to offer public methods with which clients may indirectly manipulate their values
  – An exception: constants, which are normally public static final variables
The Many Meanings of “Final”

- A **class** declared **final** may not be extended
- A **method** declared **final** may not be overridden
- A **variable** declared **final** may not be modified once it is given a value  
  – Which is why it is often called a “constant”
- A **formal parameter** declared **final** may not be modified inside the method
The Many Meanings of “Final”

- A class declared final may not be extended.
- A method declared final may not be overridden.
- A variable declared final may not be modified once it is given a value.
  - Which is why it is often called a “constant”
- A formal parameter declared final may not be modified inside the method.

Be careful! For a reference variable (or parameter):
the reference value cannot be modified, but
the object value can be modified!
Resources

• The Java Tutorials: Exceptions

• The Java Tutorials: Understanding Instance and Class Members
  – http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

• The Java Tutorials: Nested Classes
  – http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

• The Java Tutorials: Controlling Access to Members of a Class
  – http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html