Java Programming Environments

Packages

- Classes grouped into packages
- Example: `java.awt.image`
  - avoids namespace clashes
- But no semantics to having a common prefix
  - e.g., between `java.awt` and `java.awt.image`
- Package names are an implicit or explicit part of a class name

Packages (cont.)

- Import makes a class or package name implicit
  - e.g., allows use of `ColorModel` instead of `java.awt.image.ColorModel` by:
    - `import java.awt.image.ColorModel;`
  - to import all classes in a package, use *
    - e.g., `import java.awt.image.*;`
- Implicit at the beginning of every Java file
  - `import java.lang.*;`
- Import never required, just allows use of shorter names

Files – what goes where

- Each `public` class `C` must be in a file `C.java`
- If a class `C` is part of package `P`
  - `package P;` must be the first statement in `C.java`
  - which must be in a directory `P`
  - treats . in package name as subdirectories
- Reverse of domain name is reserved package name
  - `edu.umd.cs` is reserved for UMD CS department

Files (cont.)

- `CLASSPATH` gives list of places to look for class files
  - both directories and archive (jar) files
  - don’t need to specify location of system files
  - only need to set it for your own files
    - if they are part of a package
    - if they aren’t in the current directory (where the interpreter is run from)

java.lang

- Wrapper classes
- class `String`
- class `StringBuffer`
Wrapper classes

- To create `Integer`, `Boolean`, `Double`, …
  - that is a subclass of `Object`
  - useful/required for polymorphic methods
    - `Hashtable`, `LinkedList`, …
  - used in reflection classes
- Include many utility functions
  - e.g., convert to/from `String`
- `Number`: superclass of `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`
  - allows conversion to any other numeric primitive type

class String

- Cannot be changed/updated
- Automatically created for string constants
- `+` used for concatenation (arguments converted to `String` as needed)
- Lots of methods, including:
  - `int length()`, `char charAt(int pos)`
  - `int compareTo(String otherString)`
  - `void getChars(int begin, int end, char[] dst, int dstBegin)`
  - `int indexOf(int ch)`  // why doesn’t take a char??
  - `String toUpperCase()`

class StringBuffer

- StringBuffer contents can be changed
- Constructors
  - `StringBuffer()`
  - `StringBuffer(String s)`
  - `StringBuffer(int initialBufferSize)`
- Lots of methods, including
  - `StringBuffer append(String str)`
  - `StringBuffer insert(int offset, String str)`
  - both can actually take many types as argument, and convert as needed (e.g., `Object`, `int`, `float`, …)

StringBuffer Example

- Used to implement `String` concatenation

```java
String s = "(X, Y) = (" + x + ", " + y + ")";
// is compiled to:
String s = new StringBuffer("(X, Y) = (")
 .append(x).append(',').append(y).append(')').toString();
```

Exceptions and Inner Classes

- Just another class of objects
- That can be thrown
- Two subtypes
  - `Exception`
  - `Error`
  - which can always be thrown without being declared
Exception

- It is reasonable to catch and ignore exceptions
- IOException
  - all I/O errors detected by classes in java.io signaled by a subclass
- InterruptedException
  - useful for waking up sleeping or waiting threads
- RuntimeException
  - can be thrown without being declared (all standard ones are subclasses)
    - NullPointerException
    - IndexOutOfBoundsException
    - NegativeArraySizeException

Error

- Can be thrown without being declared
- Generally unreasonable to catch and ignore an error
- VirtualMachineError
  - OutOfMemoryError
  - StackOverflowError
- VerifyError
- NoClassDefFoundError

Method throws declarations

- A method declares the exceptions it might throw
  - public void openNext() throws UnknownHostException, EmptyStackException
    { ... }
- Must declare any exception the method might throw
  - unless it is caught in the method
  - includes exceptions thrown by called methods

Creating New Exceptions

- User-defined exception is just a class that is a subclass of Exception

class MyOwnException extends Exception {}
class MyClass {
  void oops() throws MyOwnException {
    if (some_error_occurred) {
      throw new MyOwnException();
    }
  }
}

Throwing an Exception/Error

- Create a new object of the appropriate Exception/Error type, and throw it
- If it’s not a subtype of Error or RuntimeException
  - must declare the method throws the exception
- Exceptions thrown are part of return type
  - when overriding a method in a superclass
  - can’t throw anything that would surprise a superclass object

Throw (cont.)

- C++ does run-time check that function doesn’t throw an unexpected exception
  - better for backward compatibility
- Java uses compile-time check
  - forces you to sometimes deal with exceptions you know can’t occur
Exception/Error Handling

• All exceptions eventually get caught
• First `catch` with supertype of the exception catches it
• Don’t catch errors
• `finally` is always executed

```java
try { if (i == 0) return; myMethod(a[i]); }
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("a[] out of bounds");
}
catch (MyOwnException e) {
    System.out.println("Caught my exception");
}
catch (Exception e) {
    System.out.println("Caught " + e.toString());
    throw e;
}
finally {
    /* stuff to do regardless of whether an exception */
    /* was thrown or a return taken */
}
```

java.lang.Throwable

• Many objects of class `Throwable` have a message
  – specified when constructed, as `String`
  – `String getMessage()` returns the message
• `String toString()`
• `void printStackTrace()`
• `void printStackTrace(PrintWriter s)`

Inner Classes

• Allow a class to be defined within a class or method
• New class has access to all variables in scope
• Classes can be anonymous
• 4 kinds of inner classes
  – nested classes/interfaces
  – standard inner classes
  – method classes
  – anonymous classes

Nested classes/interfaces

• Not really an inner class
  – not associated with an instance of the outer class
• Defined like a static class method/variable
• Can refer to all static methods/variables of outer class, transparently
• Used to localize/encapsulate classes only used by the outer class
  – information hiding/packageing
• Used to package helper classes/interfaces
  – like a mini-package for each class

Example

```java
public class LinkedList {
    // Keep this private; no one else see the implementation
    private static class Node {
        Object value; Node next;
        Node(Object v) { value = v; next = null; }
    }
    // Put here to show that this is the Transformer for LinkedList
    public static interface Transformer {
        public Object transform(Object v);
    }
    Node head, tail;
    public void applyTransformer(Transformer t) {
        for (Node n = head; n != null; n = n.next)
            n.value = t.transform(n.value);
    }
    public void append(Object v) {
        Node n = new Node(v);
        if (tail == null) head = n;
        else tail.next = n;
        tail = n;
    }
    public class(getStringRep
```