Administrivia

• C++ project due Thursday, 6PM
• Java project posted soon
  – a simple Web server
  – due Wednesday, Oct. 3, 6PM
• Java readings from *Thinking in Java*
  – we’ll do parts of many chapters, but definitely read Chapter 1 for an overview
  – I’ll add some suggestions on Web page, but use it as a reference

What Makes Java Different

• Java fully specified
  – e.g., constants, size of types, array bounds checking, no dangling pointers
  – language specification intended to completely specify the behavior of all programs
    • not just correct ones
    • all runtime errors must be caught
  – KISS principle applied
    • many useful, but not essential, features from C++ not included (operator overloading, templates, multiple inheritance, standalone functions, …)

Java semantics

• Machine architecture insensitive
  – size of machine word
  – floating point format (must use IEEE 754)
  – big/little-endian
• Compiled to machine independent byte code
• Many C++ programs break when moved to machine with different word size or endianness

Java security

• Can strictly limit access to code
  – *untrusted* code can’t access files and are limited in network connectivity by default
• Compiled code (byte codes) can be verified for correctness (by another program)
• But security bugs are still possible
  – e.g., denial of service, insecure mode code
  – but *all* C++ programs run in an insecure mode

Java features

• Strong type system
• Multi-threading and synchronization
• Garbage collection
• Exceptions
• class *Class*
  – classes are objects too
• class *Object*
  – the class all classes are derived from
Java libraries

- Utilities
  - collection classes, Zip files, internationalization
- GUIs, graphics and media
- Networking
  - sockets, URLs, RMI, CORBA
- Threads
- Databases
- Cryptography/security

Java Basics

- Mostly C++ syntax
- Statements
  - empty, expressions, blocks {}
  - control flow (if, switch, while, for, …)
  - throw and try/catch/finally
  - synchronized
  - no goto

Expressions

- Mostly C/C++ syntax
- Standard math operators: +, -, *, /, %
- Bit operations: &, |, ~, <<, >>, >>>
- Update ops: =, +=, -=, *=, /=, …
- Relational ops: <, <=, ==, >=, >, !=
- Boolean ops: &&, ||, !
- Conditional expression: b ? e1 : e2

Class operations

- Select method-variable/class/subpackage: .
- Class operators: new, instanceof, (Class)
- No pointer operations: *, &, ->

Hello World example

```java
public class HelloWorld {
    public static void main(String[] args) {
        if (args.length == 1)
            System.out.println("Hello " + args[0]);
        else
            System.out.println("Hello world");
    }
}
```

Naming conventions

- Classes/Interfaces start with a capital letter
- packages/methods/variables start with a lowercase letter
- for names with multiple words, CapitalizeFirstLetterOfEachWord
- Don’t use underscores
- CONSTANTS all in uppercase
Values

- Object reference: null, or ref to object
- boolean – a built-in type
- char – Unicode; 16 bits
- byte/short/int/long – 8/16/32/64 bits, signed
- float/double – 32/64 bits IEEE 754

Objects and references

- All objects allocated on the heap
- No object can contain another object
- All class variables/fields are references to an object
- Reference is like a C++ pointer, except
  - can only point to start of heap-allocated object
  - no pointer arithmetic allowed
  - use . instead of -> to access fields/methods

Objects and references

- All objects allocated on the heap
- No object can contain another object
- All class variables/fields are references to an object
- Reference is like a C++ pointer, except
  - can only point to start of heap-allocated object
  - no pointer arithmetic allowed
  - use . instead of -> to access fields/methods

String example

Object operations

- = assignment
  - for object references: copies reference, not object
- == equality test
  - for object references: true if references to same object
- foo.equals(bar)
  - intended to compare contents of objects
  - by default same as ==, but can/should be overridden
- foo.toString()
  - returns String representation of the object, can/should be overridden

More Object operations

- foo.clone()
  - returns shallow copy of foo (not supported on all Objects)
- foo.getClass()
  - returns class of foo (result is of type Class)
- foo instanceof Bar
  - true if object referenced by foo is a subtype of class Bar

More Object operations

- (Bar) foo
  - run-time exception if object reference by foo is not a subclass of Bar
  - compile-time error if Bar is not a subtype of foo (i.e. it always throws an exception)
  - doesn’t transform anything, just allows treating the result as if it were of type Bar
Arrays

- a special kind of object (with lots of syntax)
- can declare arrays of any type
- have one instance variable: length
- also have contents indexed with a subscript from 0 … length-1
- can be initialized using \( \{val_0, val_1, \ldots, val_n\} \) notation
  - inefficient for large arrays

Array declarations

- Little surprising for C/C++ programmer
- \texttt{int[]} \texttt{A} and \texttt{int A[]} have identical semantics
  - declares \texttt{A} to be a variable containing a reference to an array of ints
- \texttt{int[]} \texttt{A[]}, \texttt{B};
  - \texttt{A} is a ref to an array of refs to arrays of ints
  - \texttt{B} is a ref to an array of ints
- None of these allocate an array
- \texttt{A = new int [10];} allocates an array of 10 ints, and makes \texttt{A} be a reference to it

Array example

```java
int[] array1 = {1, 3, 5};
int[][] a = new int[10][3];
// a.length == 10
// a[7].length == 3
a[5][2] = 42;
a[9] = array1;
// a[9][2] == 5

// use of array initializers
int[][] twoD = {{1, 2, 3}, {4, 5}, {6}};
Object [] args = {"one", "two", a};
main(new String [] {"a", "b", "c");
```

Administrivia

- C++ project
  - private drivers posted soon
  - other student’s code will be sent to you for commentary
    - 2 projects, about a paragraph for each
- Java project posted soon
  - web server is up and running for testing
- More Java suggested readings
  - I/O – Chapter 11, pages 573-605
  - RTTI – Chapter 12
  - Distributed Computing – Chapter 15, pages 903-923

String

- A class for representing immutable strings
- string constants converted to String
- + does string concatenation
- In some contexts objects automatically converted to String type
  - Example:
    ```java
    public static void printArray(Object [] a) {
        for (int i = 0; i < a.length; i++)
            System.out.println("a[" + i + "] = " + a[i]);
    }
    ```

Object/memory allocation

- Only way/time an object gets allocated is:
  - by executing \texttt{new}
    - one object per invocation of \texttt{new}
  - by having an array constant (e.g., \texttt{\{(5, -5, 42)\}})
  - by having a string constant (e.g., "Hello world")
- Declaring a reference variable does \texttt{not} allocate an object
- Allocating an array does not automatically allocate the contents of the array
Allocation (cont.)

- Multi-dimensional array allocation
  - `int[][] a = new int[10][10];`
  - equivalent to, but faster than:
    `int[][] a = new int[10][];`
    `for(int i=0; i<10; i++) a[i] = new int[10];`
- No explicit deallocate required, nor allowed

Garbage collection

- Java uses garbage collection to find objects that cannot be referenced
  - i.e. do not have any pointers to them
- GC not a major performance bottleneck
  - fast garbage collectors have been implemented
  - on many commercial systems, GC runs on a single processor of a multiprocessor

Other notes

- Forward references resolved automatically
  - can refer to method/variable defined later
- Integer division by zero raises an exception
- Integer overflow drops extra bits
- Floating point errors create special values
  - NaN, POSITIVE_INFINITY, …
- Separate name spaces for methods, classes, variables, …
  - can produce confusing error messages

What’s missing

- preprocessor (#include, #define, …)
- structs and unions
- enumerated types
- bit-fields
- variable-length argument lists
- multiple inheritance (of implementation)
- operator overloading
- templates/ parameterized types
  - GJ (generic Java)

Java Classes

- Each object is an instance of a class
  - an array is an object
- Each class is represented by a class object
  - of type `Class`
- Each class extends one superclass
  - Object if not specified
  - except class `Object`, which has no superclass
Classes (cont.)

- Each class has methods and fields/variables
  - variables hold primitive (built-in) values or object references
- Only use '.' to access object fields
  - e.g., x.y(a.b)
- Most methods invoked using C++ virtual method semantics
  - except static, private and final methods

Class modifiers

- **public** – class visible outside package
- **final** – no other class can extend this class
- **abstract** – no instances of this class can be created
  - only instances of extensions of the class

Class details

- Method names can be overloaded
  - method invoked is determined by both its name and the types of the parameters
  - resolved at compile-time, based on compile-time types
- Methods can also be overridden
  - define a method also defined by a superclass
  - arguments and result types must be identical
  - resolved at run-time, based on type of object method is invoked on

Classes and methods

- **this** refers to the object the method is invoked on
- **super** refers to the same object as **this**
  - but used to access methods/variables in superclass
- Methods
  - can be declared in both classes and interfaces
    - only implemented in classes
  - must have a return type
    - except constructors
    - void can be used only as a return type
  - references to objects or arrays can be returned

Instance variable / method modifiers

- Visibility/access
  - **public** – visible everywhere
  - **protected** – visible within same package or in subclass
  - **package** (default) – visible within same package
- **private** – visible only within this class
- **static** – a class method or variable
Instance variable modifiers

- **transient** – not stored when object serialized
- **volatile** – don’t assume the variable hasn’t changed since the last time it was accessed
  - might be modified by another thread that doesn’t have a lock on the object
- **final** – can’t be changed; must be initialized in declaration or in constructor

Method modifiers

- **abstract** – no implementation provided
  - class must be abstract
- **final** – this method cannot be overridden
  - useful for security
  - allows compiler to inline method
- **native** – implemented in another language
- **synchronized**
  - locks object before method is executed
  - lock released after method finishes

Method arguments

- Only pass-by-value
  - but object parameters are references to heap objects that can be changed
- Only arguments used to distinguish methods
  - not return types
- Syntax same as C/C++

Overloading

- Methods with the same name, but different parameters (count or types) are overloaded

```java
class Parent {
    int cost;
    void add(int x) {
        cost += x;
    }
}
class Child extends Parent {
    void add(String s) throws NumberFormatException {
        cost += Integer.parseInt(s);
    }
}
```

Overriding

- Overriding
  - methods with same name and argument types in child class override method in parent class
  - you can override/hide instance variables
    - both variables will exist, but don’t do it

```java
class Parent {
    int cost;
    void add(int x) {
        cost += x;
    }
}
class Child extends Parent {
    void add(int x) {
        if (x > 0) cost += x;
    }
}
```

Dynamic Method Dispatch

- If you have a ref a of type A to an object that is actually of type B (a subclass of A)
  - instance methods invoked on a will get the methods for class B (like C++ virtual functions)
  - class methods invoked on a will get the methods for class A
    - invoking class methods on objects strongly discouraged
### Simple Dynamic Dispatch Example

```java
class A {
    String f() { return "A.f() " ; }
    static String g() { return "A.g() " ; }
}
public class B extends A {
    String f() { return "B.f() " ; }
    static String g() { return "B.g() " ; }
    public static void main(String args[]) {
        A a = new B();  
        B b = new B();  
        System.out.println(a.f() + a.g() + b.f() + b.g());
    }
}
```

java B generates:

```
B.f()  A.g()  B.f()  B.g()
```

### Detailed Example

- Shows
  - polymorphism for both method receiver and arguments
  - static vs. instance methods
  - overriding instance variables

#### Source code for classes

```java
class A {
    String f(A x) { return "A.f(A) " ; }
    String f(B x) { return "A.f(B) " ; }
    static String g(A x) { return "A.g(A) " ; }
    static String g(B x) { return "A.g(B) " ; }
    String h = "A.h";
    String getH() { return "A.getH(): " + h ; }
}

class B extends A {
    String f(A x) { return "B.f(A)/ " + super.f(x); }
    String f(B x) { return "B.f(B)/ " + super.f(x); }
    static String g(A x) { return "B.g(A) " ; }
    static String g(B x) { return "B.g(B) " ; }
    String h = "B.h";
    String getH() { return "B.getH(): " + h + "\" + super.h ; }
}
```

#### Invocation and results

```java
A a = new A();  A ab = new B();  B b = new B();
System.out.println( a.f(a) + a.f(ab) + a.f(b) );
System.out.println( ab.f(a) + ab.f(ab) + ab.f(b) );
System.out.println( b.f(a) + b.f(ab) + b.f(b) );
```

```
// A.f(A)  A.f(A)  A.f(B)
// B.f(A)/A.f(A)  B.f(A)/A.f(A)  B.f(B)/A.f(B)
```

```java
System.out.println( a.g(a) + a.g(ab) + a.g(b) );
System.out.println( ab.g(a) + ab.g(ab) + ab.g(b) );
System.out.println( b.g(a) + b.g(ab) + b.g(b) );
```

```
// A.g(A)  A.g(A)  A.g(B)
// B.g(A)  B.g(A)  B.g(B)
```

```java
System.out.println( a.h + a.getH() );
System.out.println( ab.h + ab.getH() );
System.out.println( b.h + b.getH() );
```

```
// A.h  A.getH():A.h
// A.b  B.getH():B.h/A.h
// B.h  B.getH():B.h/A.h
```

### What to notice

- Invoking ab.f(ab) invokes B.f(A)
  - run-time type of object determines method invoked
  - compile-time type of arguments used
- ab.h gives the A version of h
- ab.getH()
  - B.getH() method invoked
  - in B.getH(), h gives B version of h
- Use of super in class B to reach A version of methods/variables
- super not allowed in static methods

### Constructors

- Declaration syntax same as C++
  - no return type specified
  - method name same as class
- First statement can/should be this(args) or super(args)
  - if those are omitted, super() is called
  - must be very first statement, even before variable declarations
- not used for type conversions or assignments
  - void constructor generated if no constructors given
Static class components

• They belong to the class
  – static variables allocated once, no matter how many objects created
  – static methods are not specific to any class instance, so can’t refer to this or super
• Can reference class variables and methods through either class name or an object ref
  – don’t reference via object references!

Interfaces

• An interface is an object type – no associated code or instance variables
  – only describes methods supported by interface
• A class can implement (be a subtype of) many interfaces
• Interfaces may have final static variables
  – to define a set of constants (like enum in C++)

Interface example

```java
public interface Comparable {
    public int compareTo(Object o);
}
public class Util {
    public static void sort(Comparable[] arr) {
        ...;
    }
}
public class Choices implements Comparable {
    public int compareTo(Object o) {
        return ...;
    }
    ...;
    Choices[] options = ...;
    Util.sort(options);
    ...
```

No multiple inheritance

• A class type can be a subtype of many other types (implements)
• But can only inherit method implementations from one superclass (extends)
• Not a big deal
  – multiple inheritance rarely, if ever, necessary
  – and often badly used
• And it’s complicated to implement well

Garbage collection

• Objects that are no longer accessible can be garbage collected
• Method void finalize() called when an object is collected
  – don’t depend on it, since no way to tell when it will get called
• Garbage collection not a major performance bottleneck
  – new/delete in C++ can be expensive too

Class Objects

• For each class, there is an object of type Class
• Describes the class as a whole
  – used extensively in Reflection package
• Class.forName("MyClass")
  – returns class object for MyClass
  – will load MyClass if needed
• MyClass.newInstance()
  – creates a new instance of MyClass
• MyClass.class gives the Class object for MyClass
Types

- A type describes a set of values that can be:
  - held in a variable
  - returned in an expression
- Types include:
  - primitive types – boolean, char, short, int, …
  - Reference types:
    - Class types
    - Array types
    - Interface types

Class types

- Using the name of a class as a type means a reference to an instance of that class or a subclass is a permitted value
  - a subclass has all the fields of its superclass
  - a subclass has all the methods of its superclass
- null is also an allowed value

Array types

- If S is a subtype of T
  - S[] is a subtype of T[]
- Object[] is a supertype of all arrays of reference types
- Storing into an array generates a run-time check that the type stored is a subtype of the declared type of the array elements
- Performance penalty?
- Similar (and maybe worse) problems in C++

Example: Object[]

```java
public class TestArrayTypes {
    public static void reverseArray(Object[] A) {
        for(int i=0, j=A.length-1; i<j; i++, j--) {
            Object tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
        }
    }
    public static void main(String[] args) {
        reverseArray(args);
        for(int i=0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Interface types

- Using the name of an interface as a type means
  - a reference to any instance of a class that implements the interface is a permitted value
  - null is also allowed
- Object referenced is guaranteed to support all the methods of the interface
  - invoking a method on an interface might be a bit less efficient

Object Obligations

- many operations have default implementations
  - which may not be the ones you want
```java
public boolean equals(Object that) { … } // return this == that
public String toString() { … } // returns print representation
public int hashCode() { … } // key for accessing object
public void finalize() { … } // called before object garbage
public Object clone() { … } // default is shallow bit-copy if class implements Cloneable, throw CloneNotSupportedException
```

public boolean equals(Object that) { … } // return this == that
public String toString() { … } // returns print representation
public int hashCode() { … } // key for accessing object
public void finalize() { … } // called before object garbage
public Object clone() { … } // default is shallow bit-copy if class implements Cloneable, throw CloneNotSupportedException
public Object clone() { … } // default is shallow bit-copy if class implements Cloneable, throw CloneNotSupportedException

Poor man’s polymorphism

• Every object is an Object
• An Object[] can hold references to any objects
• E.g., for a data structure Set that holds a set of Object
  – can use it for a set of String
  – or a set of images
  – or a set of anything
• Java’s container classes are all containers of Object
  – when you get a value out, have to downcast it

Interacting with External Environment

Applications and I/O

• Java external interface is a public class
• via public static void main(String [] args)
• args[0] is first argument
  – unlike C/C++
• System.out and System.err are PrintStreams’s
  – should be PrintWriter’s, but would break 1.0 code
  – System.out.print(...) prints a string
  – System.out.println(...) prints a string with a newline
• System.in is an InputStream
  – not quite so easy to use

Input (JDK 1.1 and higher)

• Wrap System.in in an InputStreamReader
  – converts from bytes to characters
• Wrap the result in a BufferedReader
  – makes input operations efficient
  – supports readline() interface
• readline() returns a string
  – returns null if at EOF

Example Echo Application

import java.io.*;
public class Echo {
  public static void main(String [] args) {
    String s;
    BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
    int i = 1;
    try {
      while((s = in.readLine()) != null) {
        System.out.println((i++) + " " + s);
      }
    } catch(IOException e) {
      System.out.println(e);
    }
  }
}

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`, …
  - 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(String s1)
  - String toUpperCase()

class StringBuffer

- String 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

```
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

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

- class Exception
  - It is reasonable to catch and ignore exceptions
  - IOException
    - all I/O errors detected by classes in java.io are signaled by a subclass of IOException
  - 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
  
  ```java
  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

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

Creating New Exceptions

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

```java
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

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 error");
  throw e;
}
catch (Exception e) {
  System.out.println("Caught a raw exception");
  throw e;
} finally {
  // stuff to do regardless of whether an exception */
  // was thrown or a return taken */
}
```
java.lang.Throwable

- Methods of class `Throwable`
  - many 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 and anonymous classes
- Lots of important details

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/packaging
- 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;
    }
}
```

Standard Inner Classes

- Defined like a class method-variable
- Each instance associated with an instance of the outer class
- If class A is outer class
  - use `A.this` to get this for instance of outer class
- Can refer to all methods/variables of outer class
  - transparently
- Can’t have any static methods/variables

Example

```java
public class FixedStack {
    Object [] array;
    int top = 0;
    class MyEnum implements java.util.Enumerator {
        int count = top;
        public boolean hasMoreElements() { return count > 0; }
        public Object nextElement() {
            if (count == 0)
                throw new NoSuchElementException("FixedStack");
            return array[--count];  
        }
        public java.util.Enumerator enumerateAll() {
            return new MyEnum();  
        }
    }
} 
```

Example

```java
public class(getStringRep
```
Method and Anonymous Classes

- Can refer to all methods/variables of outer class
- Can refer to `final` local variables
- Can’t have any static methods/variables
- Method classes defined like a method variable
- Anonymous classes defined in new expression
  - `new BaseClassOrInterface() { extensions }`

Method class Example

```java
public class FixedStack {
    Object[] array;
    int top = 0;
    public java.util.Enumerator enumerateOldestK(final int k) {
        class MyEnum implements java.util.Enumerator {
            int pos = 0;
            public boolean hasMoreElements() {
                return pos < k && pos < top;
            }
            public Object nextElement() {
                if (!hasMoreElements()) {
                    throw new NoSuchElementException("FixedStack");
                    return array[pos++];
                }
            }
            return new MyEnum();
        }
        return new MyEnum();
    }
}
```

Anonymous class Example

```java
public class FixedStack {
    Object[] array;
    int top = 0;
    public java.util.Enumerator enumerateOldestK(final int k) {
        return new java.util.Enumerator() {
            int pos = 0;
            public boolean hasMoreElements() {
                return pos < k && pos < top;
            }
            public Object nextElement() {
                if (!hasMoreElements()) {
                    throw new NoSuchElementException("FixedStack");
                    return array[pos++];
                }
            }
        };
    }
}
```

Important details

- If class B is defined inside of class A
  - a synchronized method of B locks B.this, not A.this
  - may want to lock A.this for synchronization
  - can have many B’s for each A
- Can’t define constructor for anonymous inner class
- Inner classes are a compile-time transformation
  - separate class file generated for each inner class
  - have $.’s in names

Multithreading and Synchronization

What is a thread?

- It’s a program counter and a stack
- All threads share the same memory space
  - take turns with the CPU, for a uni-processor
  - run concurrently, on a multiprocessor
- Example: Web browser
  - one thread for I/O
  - one thread for each file being downloaded
  - one thread to render web page
- Running thread might
  - yield, sleep, wait for I/O or `notify`, be pre-empted
Writing Multi-threaded Code

- Need to control which events can happen simultaneously
  - e.g., update and display methods for a class
- Usually only covered in OS/DB courses
  - so few programmers have lots of training
- Can get inconsistent results or deadlock
  - problems often not easily reproduced
- Easy to get multi-threading, without trying
  - Graphical User Interfaces (GUI’s)
  - Remote Method Invocation

Extending class Thread

- Can build a thread class by extending java.lang.Thread
- Must supply a public void run() method
- Start a thread by invoking the start() method
- When a thread starts, executes run()
- When run() returns, thread is finished/dead

Simple thread methods

- void start()
- boolean isAlive()
- void setDaemon(boolean on)
  - if only daemon threads running, VM terminates
- void setPriority(int newPriority)
  - thread scheduler might respect priority
- void join() throws InterruptedException
  - waits for a thread to die/finish

Simple static thread methods

- Apply to thread invoking the method
  - void yield()
  - void sleep(long milliseconds)
    - throws InterruptedException
  - Thread currentThread()

Runnable interface

- Extending Thread means can’t extend any other class
- Instead implement Runnable
  - declares that the class has a void run() method
- Can construct a new Thread
  - and give it an object of type Runnable as an argument to the constructor
  - Thread(Runnable target)
  - Thread(Runnable target, String name)

Thread Example

```java
public class ThreadDemo implements Runnable {
    public void run() {
        for (int i = 5; i > 0; i--) {
            System.out.println(i);
            try { Thread.sleep(1000); } 
            catch(InterruptedException e) { };
        }
        System.out.println("exiting " + Thread.currentThread());
    }
    public static void main(String [] args) {
        Thread t = new Thread(new ThreadDemo(),"Demo Thread");
        System.out.println("main thread: " + t.currentThread());
        try { Thread.sleep(3000); } 
        catch (InterruptedException e) { };
        System.out.println("exiting " + Thread.currentThread());
    }
}
```
InterruptedException

- A number of thread methods throw it
  - really means: wakeUpCall
- interrupt() sends a wakeUpCall to a thread
- Won’t disturb the thread if it is working
  - but if thread attempts to sleep
  - it will get immediately woken up
- Will also wake up the thread if it is already asleep
- Thrown by sleep(), join(), wait()

Be careful with threads

- Under some implementations of JVM
  - a thread stuck in a loop will never yield by itself
- Preemptive scheduling would guarantee it
  - but not supported on all platforms
- Put yield() into loops
- I/O has highest priority, so should be able to get time on CPU

Another thread example

class UnSyncTest extends Thread {
  String msg;
  public UnSyncTest(String s) {
    msg = s;  start();  }
  public void run() {
    System.out.println("[" + msg);
    try { Thread.sleep(1000); } catch(InterruptedException e) {} System.out.println("]");
  }
  public static void main(String [] args) {
    new UnSyncTest("Hello");
    new UnSyncTest("UnSynchronized");
    new UnSyncTest("World");
  }
}

Daemon threads

- A thread can be marked as a daemon thread
- By default, thread acquires status of thread that spawned it
- When no threads running except daemons
  - program execution terminates

Synchronization issues

- Locks
- synchronized statements and methods
- wait and notify
- Deadlock

Locks

- All objects can be locked
- Only one thread can hold a lock on an object
  - other threads block until they can acquire it
- If your thread already holds a lock on an object
  - can lock it a second time
  - object not unlocked until both locks released
- No way to only attempt to acquire a lock
Synchronized methods

- A method can be synchronized
  - add synchronized modifier before return type
-Obtains a lock on object referenced by this, before executing method
  - releases lock when method completes
-For a static synchronized method
  - locks the class object

Synchronized statement

- synchronized (obj) { statements }
- Obtains a lock on obj before executing statements in block
- Releases lock once block completes
- Provides finer grained control than synchronized method
- Allows locking arguments to a method

Synchronization example

class SyncTest extends Thread {
    String msg;
    public SyncTest(String s) {
        msg = s;
        start();
    }
    public void run() {
        synchronized (SyncTest.class) {
            System.out.print("[");
            try { Thread.sleep(1000); } catch (InterruptedException e) {};
            System.out.println("]");
        }
    }
    public static void main(String [] args) {
        new SyncTest("Hello");
        new SyncTest("Synchronized");
        new SyncTest("World");
    }
}

Wait and Notify

- Must be called inside synchronized method or block of statements
- a.wait()
  - gives up lock(s) on a
  - adds thread to wait set for a
  - suspends thread
- a.wait(int m)
  - limits suspension to m milliseconds

Wait and Notify (cont.)

- a.notify() resumes one thread from a’s wait list
  - and removes it from wait set
  - no control over which thread
- a.notifyAll() resumes all threads on a’s wait list
- resumed threads must reacquire lock before continuing
- wait doesn’t give up locks on any other objects
  - e.g., acquired by methods that called this one

Producer/Consumer Example

public class ProducerConsumer {
    private boolean ready = false;
    private Object obj;
    public ProducerConsumer() { }
    public ProducerConsumer(Object o) {
        obj = o; ready = true; 
    }
    synchronized void producer(Object o) {
        synchronized Object consumer() {
            synchronized void producer(Object o) {
                while (ready) wait();
                obj = o; ready = true;
                notifyAll(); 
                return obj; 
            }
        }
    }
    synchronized Object consumer() {
        synchronized void producer(Object o) {
            while (!ready) wait();
            ready = false;
            notifyAll();
        }
    }
}
**Changed example**

synchronized void produce(Object o) {
    while (ready) {
        wait();
        if (ready) notify();
    }
    obj = o;  ready = true;
    notify();
}

Bad design – no guarantee about who will get woken up

**A Better Solution**

synchronized void produce(Object o) {
    while (ready) { synchronized (empty) {
        try {empty.wait(); }
        catch (InterruptedException e) { }
    }
    obj = o;  ready = true;
    synchronized (full) {
        full.notify();
    }
}

synchronized Object consume() {
    synchronized (full) {
        full.notify();
    }
    return obj;
}

Use two objects, empty and full, to allow two different wait sets

**notify() vs. notifyAll()**

- Very tricky to use notify() correctly
  - notifyAll() much safer
- Need:
  - all waiters are equal
  - each notify only needs to wake up 1 thread
  - handle InterruptedException correctly

**InterruptedException Example**

- Threads t1 and t2 are waiting
- Thread t3 performs a notify
  - thread t1 is selected
- Before t1 can acquire lock, t1 is interrupted
- t1’s call to wait throws InterruptedException
  - t1 doesn’t process notification
  - t2 doesn’t wake up

**Handling InterruptedException**

synchronized (this) {
    while (!ready) {
        try { wait(); } catch (InterruptedException e) {
            notify();
            throw e; }
        // do whatever
    }
}

**Deadlock**

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on A
  - Thread 2 holds lock on B
  - Thread 1 is trying to acquire a lock on B
  - Thread 2 is trying to acquire a lock on A
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - other than by the fact that nothing is happening
**A common multi-threading bug**

- Threads might cache values
- Obtaining a lock forces the thread to get fresh values
- Releasing a lock forces the thread to flush out all pending writes
- `volatile` variables are never cached
- `sleep(...)` doesn’t force fresh values
- Many compilers don’t perform these optimizations — but some do (Hotspot?)
- Problem might also occur with multiple CPUs

**Guidelines to simple/safe multi-threaded programming**

- Synchronize access to shared data
- Don’t hold a lock on more than one object at a time
  - could cause deadlock
- Hold a lock for as little time as possible
  - reduces blocking waiting for locks
- While holding a lock, don’t call a method you don’t understand
  - e.g., a method provided by someone else, especially if you can’t be sure what it locks

**Guidelines (cont.)**

- Have to go beyond these guidelines for more complex situations
  - but need to understand threading and synchronization well
- Recommended book for more details:
  - *Concurrent Programming in Java*, by Doug Lea

**I/O and Utility Libraries**

**I/O Classes**

- File
  - directories
    - `if(isDirectory()) System.out.println(f.list());`
  - interface `FilenameFilter` — allows selection of sublist
- `OutputStream` — byte stream going out
- `Writer` — character stream going out
- `InputStream` — byte stream coming in
- `Reader` — character stream coming in

**OutputStream - bytes**

- base types
  - `ByteArrayOutputStream`
  - `FileOutputStream` — goes to file
  - `PipedOutputStream` — goes to `PipedInputStream`
  - `SocketOutputStream` (not public) — goes to TCP socket
- Filters — wrapped around an `OutputStream`
  - `BufferedOutputStream`
  - `ObjectOutputStream` (should implement `FilterOutputStream`) — serialization of object graph
### Writer - characters
- **OutputStreamWriter**
  - wraps around OutputStream to get a Writer
  - takes characters, converts to bytes
  - can specify encoding used to convert
- **CharArrayWriter**
- **StringWriter**
- **Filters**
  - PrintWriter – supports println
  - BufferedWriter
- **Convenience writers**
  - wrap OutputStreamWriter around an OutputStream
  - FileWriter and PipedWriter

### InputStream - bytes
- base types
  - ByteArrayInputStream
  - FileInputStream
  - PipedInputStream
  - SocketInputStream (not public) – comes from TCP socket
- **Filters** – wrapped around InputStream
  - BufferedReader
  - PushedBackInputStream
  - ObjectInputStream
- **SequenceInputStream** - concatenate

### Reader - characters
- **InputStreamReader**
  - wrap around InputStream to get a Reader
  - takes bytes, converts to characters
  - can specify encoding used to convert
- **CharArrayReader**
- **StringReader**
- **Filters**
  - BufferedReader – efficient, supports readLine()
    - LineNumberReader – reports line numbers
  - PushBackReader
- **Convenience Readers**
  - wrap InputStreamReader around InputStream
  - FileReader and PipedReader

### java.util
- **Vector**
- **Dictionaries**
- **Enumerations and Bitsets**
- **Collection classes**

### Vector
- A list/vector abstraction
- Can insert/delete/modify elements anywhere in the list
- Can access by position
- Stack
  - extension of Vector
  - adds push, pop, peek and empty

### Dictionaries
- **Dictionary**
  - an abstract class
  - represents key to value mapping
- **HashTable**
  - an implementation of Dictionary
  - grows as needed
  - can be saved to a file (serializable, implements toString)
Enumerations and Bitsets

• Enumeration
  – an interface
  – used in many places to return an enumeration
    • public boolean hasMoreElements()
    • public Object nextElement()

• BitSet
  – provides representation of a set as a bit vector
  – grows as needed (like HashTable)

Collection Classes

• interface Collection
  – interface List
    • class Vector (and Stack)
    • class ArrayList
      – class LinkedList
      – doubly linked
  – interface Set
    • class HashSet
    • interface SortedSet
      – class TreeSet
  – interface Map – dictionary-like structures
    • class HashMap – replaces HashTable
    • interface SortedMap
      • class TreeMap

Other libraries

• java.lang.Math
  – abstract final class – only static members
  – includes constants $e$ and $\pi$
  – includes static methods for trig, exponentiation, min, max, …

• java.text
  – text formatting tools
    • class MessageFormat provides printf/scanf functionality
  – lots of facilities for internationalization