CS11 – Java

Winter 2014-2015

Lecture 2

Today's Topics

- Packages
- Interfaces
- Collection classes

Java Packages

- Classes can be grouped into <u>packages</u>
  - A package is a collection of related types
- Packages provide namespace management
  - Can't have two classes with same name in same package
  - Classes can have the same name, if they are in *different* packages
- By default, a class is in the "default package"
  - Default package has no name!
- Use **`package`** keyword to specify different package

```
package cs11;
```

  - Must be first statement in your **`.java`** file
  - Affects where **`.java`** and **`.class`** files must be placed!
  - For now, don't specify **`package`** keyword for your classes

Using Classes in Packages

- If a class is in a package, one of three choices:
  - Must refer to class with *qualified* name:

```
java.util.ArrayList myList =
  new java.util.ArrayList();
```

  - Must *import* the class itself:

```
import java.util.ArrayList;
...
ArrayList myList = new ArrayList();
```

  - Must import entire package:

```
import java.util.*;
...
ArrayList myList = new ArrayList();
```

The Java API and Packages

- All Java API classes are in packages
- Classes in **`java.lang`** are automatically imported
  - Don't need to explicitly import anything in **`java.lang`**
- To import Java classes not in **`java.lang`** package:

```
import java.util.ArrayList;
import java.util.HashSet;
...
```

  - Or:

```
import java.util.*;
```

- Importing a package is *not* recursive!
  - Importing **`java.*`** won't get you anywhere.

Sets of Behaviors

- Frequently have situations where:
  - A single, well-defined set of behaviors…
  - …with *many* different possible implementations
- Interfaces are similar to classes, but only contain method signatures with no bodies
  - They only *declare* behavior; they don't *define* it
  - No method implementations, no instance fields
- A class can implement multiple interfaces
  - Called multiple interface inheritance in Java
  - (Java doesn't support multiple class inheritance)

Interfaces

- Interfaces “define a protocol of communication between two objects.”
  - The interface declares a set of methods (behaviors)
- A class implements an interface to denote that it provides that set of behaviors
- Code other objects against the <u>interface</u> type
  - Isolates them from the implementation details specific to the implementing object

Declaring Interfaces

- Interfaces are declared like classes

```java
/** A generic component of a simulation. */
public interface SimComponent {
    /** Initialize the component. */
    void init(SimConfig sconf);

    /** Advance the simulation. */
    void simulate(double timestep);

    /** End the simulation. */
    void shutdown();
}
```

  - Goes in **`SimComponent.java`**
  - No method access-modifiers! Access is <u>public</u>.

Interfaces and Classes

- Classes can implement interfaces
  - Allows instances to be treated as the interface type
  - A class can implement any number of interfaces
  - A simpler, cleaner version of multiple inheritance
- Interfaces themselves cannot be instantiated
  - They must be implemented by a class, and then the class is instantiated
- Variables *can* be an interface type, just like they can be a class type

Implementing Interfaces

- When a class implements the interface, it must declare the methods as <u>public</u>.

```
public class PhysicsEngine implements SimComponent {
    ...
    public void init(SimConfig simConf) {
        ...  // Do some stuff
    }
    public void simulate(double timestep) {
        ...  // Do other stuff
    }
    ...
}
```

  - <u>Anyone</u> can call the class' implementation of interface, because it's public.

Using Interfaces

- Use interfaces to decouple program components
  - …especially when a component may be implemented in multiple ways!
  - Other components interact with the general interface type, not specific implementations
- Example: storing a user's calendar of events

```
public interface CalendarStorage {
    // Load a user's calendar of events
    Calendar loadCalendar(String username);

    // Save the user's calendar to persistent storage
    void saveCalendar(String username, Calendar c);
}
```

Using Interfaces (2)

- Provide multiple implementations
  - Store calendars in local data files:

```
public class FileCalendarStorage
    implements CalendarStorage {
    ...
}
```

  - Store calendars on a remote server:

```
public class RemoteCalendarStorage
    implements CalendarStorage {
    ...
}
```

- Write code to the interface, not implementations

```
CalendarStorage calStore = openCalendarStorage();
Calendar cal = calStore.loadCalendar(username);
```

Using Interfaces (3)

- Can change implementation details as needed…
  - …as long as interface definition *stays the same.*
- If interface's implementation is large and complex:
  - Other code can use a "stubbed-out" implementation of the interface, until the full version is finished

```
public class FakeCalendarStorage
        implements CalendarStorage {
        public Calendar loadCalendar(String username) {
            return new Calendar(username);  // Blank calendar
        }
        public void saveCalendar(String username,Calendar c) {
            // Do nothing!
        }
}
```

  - Allows software development of dependent components to proceed in parallel

Extending Interfaces

- Can extend interfaces:

```
/** A sim-component that runs in a network. */
public interface DistributedSimComponent
    extends SimComponent {

    /** Establish connection to server. */
    void connect(String hostname);

    /** Disconnect from server. */
    void disconnect();
}
```

  - This interface inherits all **`SimComponent`** method declarations
  - Again, they are all public access

Java Collections

- Very powerful set of classes for managing collections of objects
  - Introduced in Java 1.2
- Provides:
  - Interfaces specifying different kinds of collections
  - Implementations with different characteristics
  - Iterators for traversing a collection’s contents
  - Some common algorithms for collections
- Very useful, but nowhere near the power and flexibility of C++ STL

Why Provide Collection Classes?

- Reduces programming effort
  - Most programs need collections of some sort
  - Makes language more appealing for development
- Standardized interfaces and features
  - Reduces learning requirements
  - Facilitates interoperability between separate APIs
- Facilitates fast and correct programs
  - Java API provides high-performance, efficient, correct implementations for programmers to use

Collection Interfaces

- Generic collection interfaces defined in **`java.util`**
  - Defines basic functionality for each kind of collection
- **`Collection`** – generic "bag of objects"
- **`List`** – linear sequence of items, accessed by index
- **`Queue`** – linear sequence of items "for processing"
  - Can add an item to the queue
  - Can "get the next item" from the queue
  - What is "next" depends on queue implementation
- **`Set`** – a collection with no duplicate elements
- **`Map`** – associates values with unique keys

More Collection Interfaces

- A few more collection interfaces:
  - **`SortedSet`** (extends **`Set`**)
  - **`SortedMap`** (extends **`Map`**)
  - These guarantee iteration over elements in a particular order
- Requires elements to be comparable
  - Must be able to say an element is “less than” or “greater than” another element
  - Provide a total ordering of elements used with the collection

Common Collection Operations

- Collections typically provide these operations:
  - `add(Object o)` – add an object to the collection
  - `remove(Object o)` – remove the object
  - `clear()` – remove all objects from collection
  - `size()` – returns a count of objects in collection
  - `isEmpty()` – returns true if collection is empty
  - `iterator()` – traverse contents of collection
- Some operations are optional
  - Throws `UnsupportedOperationException` if not supported by a specific implementation
- Some operations are slower/faster

Collection Implementations

- Multiple implementations of each interface
  - All provide same basic functionality
  - Different storage requirements
  - Different performance characteristics
  - Sometimes other enhancements too
    - e.g. additional operations not part of the interface
- Java API Documentation gives the details!
  - See interface API Docs for list of implementers
  - Read API Docs of implementations for performance and storage details

`List` Implementations

- **`LinkedList`** – doubly-linked list
  - Each node has reference to previous and next nodes
  - O(N)-time element indexing
  - Constant-time append/prepend/insert
  - Nodes use extra space (previous/next references, etc.)
  - Best for when list grows/shrinks frequently over time
  - Has extra functions for get/remove first/last elements
- **`ArrayList`** – stores elements in an array
  - Constant-time element indexing
  - Append is usually constant-time
  - O(N)-time prepend/insert
  - Best for when list doesn't change much over time
  - Has extra functions for turning into a simple array

`Set` Implementations

- **`HashSet`**
  - Elements are grouped into “buckets” based on a hash code
  - Constant-time add/remove operations
  - Constant-time “contains” test
  - Elements are stored in no particular order
  - Elements must provide a hash function
- **`TreeSet`**
  - Elements are kept in sorted order
    - Stored internally in a balanced tree
  - O(log(N))-time add/remove operations
  - O(log(N))-time “contains” test
  - Elements must be comparable

`Map` Implementations

- Very similar to **`Set`** implementations
  - These are *associative containers*
  - Keys are used to access values stored in maps
  - Each key appears <u>only once</u>
    - (No multiset/multimap support in Java collections)
- **`HashMap`**
  - Keys are hashed
  - Fast lookups, but random ordering
- **`TreeMap`**
  - Keys are sorted
  - Slower lookups, but kept in sorted order by key

Collections and Java 1.5 Generics

- Up to Java 1.4, collections only stored **`Objects`**

```
LinkedList points = new LinkedList();
points.add(new Point(3, 5));
Point p = (Point) points.get(0);
```

  - Casting everything gets annoying
  - Could add non-**`Point`** objects to **`points`** collection too!
- Java 1.5 introduces generics
  - A class or interface can take other types as parameters

```
LinkedList<Point> points = new LinkedList<Point>();
points.add(new Point(3, 5));
Point p = points.get(0);
```

  - No more need for casting
  - Can only add **`Point`** objects to **`points`** too
  - Syntactic sugar, but quite useful!

Using Collections

- Lists and sets are easy:

```
HashSet<String> wordList = new HashSet<String>();
LinkedList<Point> waypoints = new LinkedList<Point>();
```

  - Element type must appear in both variable declaration and in **`new`**-expression
- Maps are more verbose:

```
TreeMap<String, WordDefinition> dictionary =
    new TreeMap<String, WordDefinition>();
```

  - First type is key type, second is the value type
- Java 7 introduces a simplified syntax:

```
TreeMap<String,WordDefinition> dictionary = new TreeMap<>();
```

  - Parameters for instantiation are inferred from variable

Iteration Over Collections

- Often want to iterate over values in collection
- **`ArrayList`** collections are easy:

```
ArrayList<String> quotes;
...
for (int i = 0; i < quotes.size(); i++)
  System.out.println(quotes.get(i));
```

  - Impossible/undesirable for other collections!
- Iterators are used to traverse contents
- **`Iterator`** is another simple interface:
  - **`hasNext()`** – Returns **`true`** if can call **`next()`**
  - **`next()`** – Returns next element in the collection
- **`ListIterator`** extends **`Iterator`**
  - Provides many additional features over **`Iterator`**

Using Iterators

- Collections provide an **`iterator()`** method
  - Returns an iterator for traversing the collection
- Example:

```
HashSet<Player> players;
...
Iterator<Player> iter = players.iterator();
while (iter.hasNext()) {
  Player p = iter.next();
  ... // Do something with p
}
```

  - Iterators also use generics
  - Can use iterator to delete current element, etc.

Java 1.5 Enhanced For-Loop Syntax

- Setting up and using an iterator is annoying
- Java 1.5 introduced syntactic sugar for this:

```
for (Player p : players) {
    ... // Do something with p
}
```

  - Can't access the actual iterator used in the loop
  - Best for simple scans over a collection's contents
- Can also use enhanced for-loop syntax with arrays:

```
float sum(float[] values) {
    float result = 0.0f;
    for (float val : values)
        result += val;
    return result;
}
```

Collection Algorithms

- **`java.util.Collections`** class provides *some* common algorithms
  - …not to be confused with the **`Collection`** interface
  - Algorithms are provided as static functions
  - Implementations are fast, efficient, and generic
- Example: sorting

```
LinkedList<Product> groceries;
...
Collections.sort(groceries);
```

  - Collection is sorted <u>in-place</u>: **`groceries`** is changed
- Read Java API Docs for more details
  - Also see **`Arrays`** class for array algorithms

Collection Elements

- Collection elements may require certain capabilities
- **`List`** elements don't need anything special
  - …unless **`contains()`**, **`remove()`**, etc. are used!
  - Then, elements should provide a correct **`equals()`** implementation
- Requirements for **`equals()`**:
  - **`a.equals(a)`** returns true
  - **`a.equals(b)`** same as **`b.equals(a)`**
  - If **`a.equals(b)`** is true and **`b.equals(c)`** is true, then **`a.equals(c)`** is also true
  - **`a.equals(null)`** returns false

Set Elements, Map Keys

- Sets and maps require special features
  - Sets require these operations on set-elements
  - Maps require these operations on the keys
- **`equals()`** must definitely work correctly
- **`TreeSet`**, **`TreeMap`** require sorting capability
  - Element or key class must implement **`java.lang.Comparable`** interface
  - Or, an appropriate implementation of **`java.util.Comparator`** must be provided
- **`HashSet`**, **`HashMap`** require hashing capability
  - Element or key class must provide a good implementation of **`Object.hashCode()`**

`Object.hashCode()`

- **`java.lang.Object`** has a **`hashCode()`** method

  ```
  public int hashCode()
  ```

  - Compute a hash code based on object's values
  - **`hashCode()`** is used by **`HashSet`**, **`HashMap`**, etc.
- Rule 1:
  - If **`a.equals(b)`** then their hash codes must be the same!
  - OK for two non-equal objects to have the same hash code
    - "Same hash-codes" just means "they *might be* equal"
- Rule 2:
  - If you override **`equals()`** on a class then you should also override **`hashCode()`**!
  - (See Rule 1)

Implementing **`hashCode()`**

- Is this a correct implementation?

```
public int hashCode() {
    return 42;
}
```

  - It satisfies the rules, so *technically* yes…
  - In practice, will cause programs to be <u>very</u> inefficient
- Hash fn. should generate a wide range of values
  - Specifically, should produce a uniform distribution of values
  - Facilitates most efficient operation of hash tables
  - <u>Requirement</u> is that equal objects must produce identical hash values…
  - Also good if unequal objects produce different hash values

Implementing **`hashCode()`** (2)

- If a field is included in **`equals()`** comparison, should also include it in the hash code
- Combine individual values into a hash code:

```
int hashCode() {
    int result = 17;     // Some prime value

    // Use another prime value to combine
    result = 37 * result + field1.hashCode();
    result = 37 * result + field2.hashCode();
    ...
    return result;
}
```

More Hash-Code Hints

- A few more basic hints:
  - If field is a boolean, use 0 or 1 for hash code
  - If field is an integer type, cast value to **`int`**
  - If field is a non-array object type:
    - Call the object's hashCode() function, or use 0 for **`null`**
  - If field is an array:
    - Include every array-element into final hash value!
    - (Arrays already do this for you – see prev. point)
  - See <u>Effective Java</u>, Item 8 for more guidelines!
- If computing the hash is expensive, cache it.
  - Must recompute hash value if object changes!

Comparing and Ordering Objects

- Objects implement **`java.lang.Comparable<T>`** interface to allow them to be ordered

  ```
  public int compareTo(T obj)
  ```
- Returns a value that imposes an order:
  - result < 0 means **`this`** is less than **`obj`**
  - result == 0 means **`this`** is "same as" **`obj`**
  - result > 0 means **`this`** is greater than **`obj`**
- This defines the *natural ordering* of a class
  - i.e. the "usual" or "most reasonable" sort-order
- Natural ordering should be *consistent with* **`equals()`**
  - **`a.compareTo(b)`** returns 0 only when **`a.equals(b)`** is true
- Implement this interface correctly for using **`TreeSet`** / **`TreeMap`**

Alternate Orderings

- Can provide extra comparison functions
  - Provide a separate object that implements **`java.util.Comparator<T>`** interface
  - Simple interface:

    ```
    int compare(T o1, T o2)
    ```
- Sorted collections, sort algorithms can also take a comparator object
  - Allows sorting by all kinds of things!
- Comparator impls are typically nested classes
  - e.g. **`Player`** class could provide a **`ScoreComparator`** nested class

Implmenting Interfaces with Generics

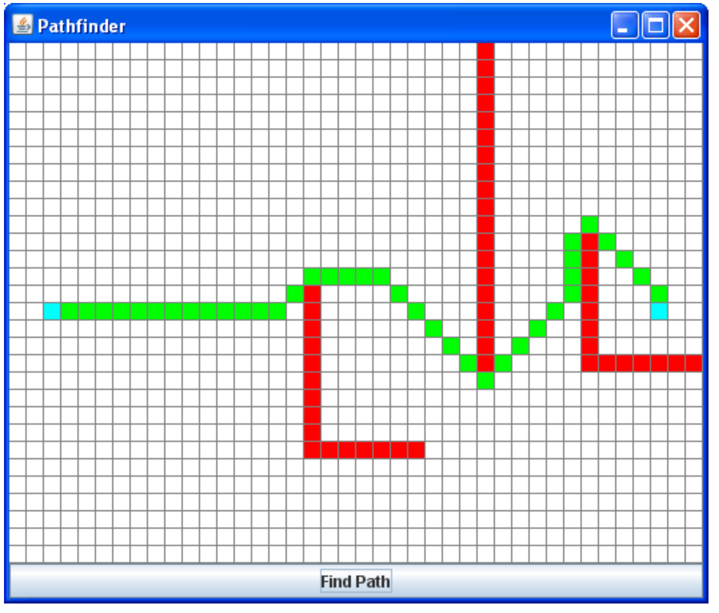
- Java interface type: **`java.lang.Comparable<T>`**
  - **`int compareTo(T obj)`**
- When you implement interfaces like this, you specify what **`T`** is in your code:

```
class Player implements Comparable<Player> {
    ...
    int compareTo(Player obj) {
        ...
    }
}
```

- Similar approach with **`java.util.Comparator`**

Lab 2 – A* Path-Finding Algorithm

- A* path-finding algorithm is used extensively for navigating maps with obstacles
  - Finds an optimal path from start to finish, if a path exists
- Example:



A* Implementation

- A* algorithm requires two collections
  - A collection of “open waypoints” to be considered
  - Another collection of “closed waypoints” that have already been examined
- Your tasks:
  - Provide **`equals()`** and **`hashCode()`** impls. for **`Location`** class
  - Complete the **`AStarState`** class, which manages open and closed waypoints for A* algorithm
  - Play with the fun A* user interface ☺