CSE 12: Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Six
9 Aug 2011
Circular linked lists.
Circular linked lists

- Before moving on to other data structures, we will discuss one more variant of the basic “linked list” concept.
- A *circular linked list* is a list where the tail’s “next” pointer points back to the *head*.
- If the linked list is doubly-linked, then the head’s “previous” pointer also points back to the *tail*.
Circular linked lists
Circular linked lists

With a circular linked list, we don’t even really need a tail.

Instead, all we really care about is whether we add to the front of the list (to the “right” of _head), or to the back of the list (to the “left” of _head).
Circular linked lists

- The utility of circular linked lists is perhaps most clearly illustrated when there are no dummy nodes.
- Empty list: `_head = null`.
- List of size 1:
Circular linked lists

- List of size 2:
Iterating through a circular linked list

- As long as a circular linked list is non-empty, an Iterator can iterate **forever**.

- Just keep following the current Node's \_next pointer.

```java
class CircularListIterator {
    Node _current;
    ...
    boolean hasNext () {
        return _listSize > 0;
    }
    Object next () {
        _current = _current._next;
        return _current._data;
    }
}
```
Simulating a circular linked list

• Using `DoublyLinkedList12` (with dummy nodes, but without pointers to “loop back around”), we can easily simulate a circular linked list.

• In `Iterator.next()`, if we’ve iterated to the tail, then just start back over at the head...

```java
Object next () {
    if (_current == _tail) {  // Loop back
        _current = _head;
    }
    _current = _current._next;
    ...
}
```
Circular linked lists

• Circular linked lists are most useful for storing a collection of objects in which “looping forever” is an intuitive and useful operation.

• Examples:
  • Looping around vertices of a polygon.
Circular linked lists

- CPU scheduling:
  - One CPU can only execute one computer program at any given time.
  - On a single-core machine, to simulate "multitasking", each program is given a small "timeslice" (few milliseconds) to run on the CPU.
  - After the timeslice expires, the next program in the list of processes is selected, and so on.
  - After all programs in the list have received their timeslice, the CPU scheduler goes back to the first process.
Circular linked lists for CPU scheduling

Circular linked list of processes waiting for CPU access.

CPU Scheduler _current
Type-safety and casting.
Type-safety in Java

• As mentioned in Lecture Three, Java was designed from the ground up to offer security.

• One aspect of security is ensuring that a variable that, for example, is supposed to point to a String doesn’t actually point to an Integer.

    // Won’t compile
    final String s = new Integer(6);

• This form of security is known as type-safety.
That example was somewhat obvious; let’s look at a more subtle one...

```java
definal Object o = new Integer();
definal String s = (String) o;
```

This code will compile ok, ...
That example was somewhat obvious; let's look at a more subtle one...

```java
final Object o = new Integer();
final String s = (String) o;  // Compiles ok
```

...but at run-time, the second statement will trigger a `ClassCastException` -- an `Integer` is never also a `String`!
Type-safety

• Java and the JVM enforce type-safety:
  • Every object knows what kind of class it is, what its parent class is, and all the interfaces that it implements.
  • If you attempt to “cast” an object into a type with which it is not compatible, then this will trigger a ClassCastException.
• Your program will terminate.
• In object-oriented languages like Java, objects are cast into different classes/interfaces when we assign them to reference variables of different types.

• Consider:

```java
class A {
    ...
}
class B extends A {
    ...
}

B b = new B();
A a = b;       // Upcast from B to A.
B b2 = (B) a;  // Downcast from A to B.
```

The terms upcast and downcast have to do with the class hierarchy, in which parent classes are “above” child classes.
Upcasting

• If class B is a subclass of A, and we convert a reference of type B to a reference of type A, then we are upcasting, e.g.: A a = b;

• Since all objects of type B are implicitly also of type A, this cast is guaranteed to succeed.

• Every object of type B can also be treated as an object of type A.

• All methods and instance variables of A are guaranteed to be accessible.
Downcasting

• If class \( B \) is a subclass of \( A \), and we convert a reference of type \( A \) to a reference of type \( B \), then we are downcasting, e.g.: \( B \ b = (B) \ a \);

• Since objects of type \( A \) are not guaranteed to always also be of type \( B \), we must explicitly inform the compiler that we “know” that \( b \) is of class \( B \).

• We must explicitly “request” the cast by writing \((B)\).
Downcasting

- At run-time, before performing the cast from class `A` to `B`, the JVM will check whether `B` is actually a `B` object.
  - If it is, then execution proceeds merrily.
  - If not, then the JVM will throw a `ClassCastException`.

Tuesday, August 9, 2011
Casting to interfaces

- We can also cast to an interface type, e.g.:
  
  ```java
  Object o = new DoublyLinkedList12();
  Iterable iterable = (Iterable) o;
  ```

- Since not every object of `Object` class is guaranteed to implement the `Iterable` interface, we must “downcast” to `Iterable`.

- At run-time, the JVM will check whether `o` is of some class that implements `Iterable`, and throw a `ClassCastException` if it is not.
Importance of type-safety

- Not all languages are type-safe.
- In C++, for example, the compiler is happy to compile the following code:

  Here, we “force” the compiler to treat the Integer pointer as a Student pointer.

  ```
  Integer *integer = new Integer(123);
  Student *student = (Student *) integer;
  student->_age = 23;
  
  Here we attempt to modify the _age instance variable of a “Student” object. But student actually points to an Integer object!
  ```
Danger in casting

• The outcome of this program can’t be good -- we’re trying to modify the “_age” of an Integer object!

• What’s going on here in terms of memory?

• Let’s first convert this example to Java...
Danger in casting

- Let’s assume the following class definitions:

```java
class Integer {  // 4 bytes total
    int _value;
}

class Student {  // 8 bytes total
    String _name;
    int _age;
}
```
Danger in casting

Integer integer = new Integer(123);
Student student = (Student) integer;

Address | Contents
---|---
8192 | integer: 8200
8196 | student: 8200
8200 | _value: 123
8204 | Something important
Danger in casting

```
Integer integer = new Integer(123);
Student student = (Student) integer;
```

Let's also suppose a “real” Student object would look like this:

![Diagram of memory addresses and object structure]
Danger in casting

Integer integer = new Integer(123);
Student student = (Student) integer;
student._age = 85;

• In the last line of code, the program attempts to modify the "_age" instance variable of the "Student" object pointed to by student.

• _age would be stored at 8204.

• In reality, student actually points to an Integer object.

• That Integer object does not own address 8204!

• Something Important has been clobbered.
Clobbering memory

• When you write data outside of a variable’s “proper bounds”, you are “clobbering memory”.

• In the previous example, treating the Integer like a Student caused the statement student._age = 85 to overwrite Something Important.

• Without Java’s protective type safety, this could cause your program to:
  • Crash.
  • Behave in unexpected ways at some indeterminate point in the future. \(\Leftarrow\text{ Often worse than crashing.}\)
Clobbering memory

• In some settings (e.g., a web server application that processes data sent from user), treating a variable as an object of the wrong type could be exploited by an attacker.

• By causing your code to “clobber” the right memory, an attacker might gain control of your entire machine. :-(
Type-safety in Java

• Java and the JVM help to prevent such attacks.

• All downcasts are checked by the JVM to make sure they are valid before execution proceeds.

• As always, this added security comes at a cost:
  • A downcast in Java is slower than a downcast in C++.
Java collections before generics.
Java Collections Framework

• Since Java version 1.2, the JDK has offered pre-built “collections” of various types as part of the Java Collections Framework (JCF).

• The JCF includes such classes as:
  • ArrayList
  • Vector
  • HashTree
  • Set
  • etc., etc.
CSE12 Collections

• In this course, we have worked on two “collections” -- ArrayList, and DoublyLinkedList12.

• Similar to the JCF collections in Java 1.2, our collections have dealt with Objects:
  • public void add (Object o);
  • public Object get (int index);

• Every object in Java is of type Object; hence, these collections can store variables of any type.
Collections of Objects

• Hence, the same class `ArrayList` can be used to create a list of `Strings` as well as a list of `Integers`:

```java
final ArrayList listOfStrings = new ArrayList();
listOfStrings.add("yo");

final ArrayList listOfIntegers = new ArrayList();
listOfIntegers.add(new Integer(32));
```

• This is convenient -- we don’t have to create a two different classes to store `Strings versus Integers`.
Downside of downcasting

• Unfortunately, the fact that the List12 interface takes and returns objects also means that we have to *downcast* the object every time we call `get(index)`:

```java
list.add("hello");
final String s = (String) listOfStrings.get(0);
```

• Having to *downcast* every time is both *tedious* and *distracting* because it litters the code with parentheses and class names.
There's also a security reason why downcasting an `Object` returned by a collection is bad:

- We may accidentally try to downcast an `Object` to an incompatible type.
Consider a method in which you use several collections to store data of several types:

```java
ArrayList list1, list2, list3;
list1 = new ArrayList(); // for Strings
list2 = new ArrayList(); // for Integers
list3 = new ArrayList(); // for Students
list1.add("test");
list2.add(new Integer(17));
list2.add(new Integer(42));
...
list3.add(new Student());
list1.add(new Student());
list2.add(new Integer(4));
list1.add("another string");
```
Downside of downcasting

- Consider a method in which you use several collections to store data of several types:

```java
ArrayList list1, list2, list3;
list1 = new ArrayList();  // for Strings
list2 = new ArrayList();  // for Integers
list3 = new ArrayList();  // for Students
list1.add("test");
list2.add(new Integer(17));
list2.add(new Integer(42));
...
list3.add(new Student());
list1.add(new Student());  // Wrong list!
list2.add(new Integer(4));
list1.add("another string");
```
Downside of downcasting

- If we later retrieve an object from list1 and assume (incorrectly) that it contains only Strings, our program will crash:

```java
final Iterator iterator = list1.iterator();
while (iterator.hasNext()) {
    final String s = (String) iterator.next();
    ...
}  // Given the code on previous slide, this will trigger a ClassCastException.
```

- It is still nice that the JVM catches our mistake at run-time, but it would be even nicer for the Java compiler to catch our mistake at compile-time.
Downside of downcasting

- Unfortunately, with collections of objects, this is not really possible.

- The compiler has no way of “knowing” that list1 was intended “only for Strings”.

- `ArrayList.add(o)` is happy to accept any `Object o`.

Tuesday, August 9, 2011
More plausible example

• A more plausible example of the problem above might occur if you are implementing a method that takes a collection as a parameter:

```java
// Specified list should contain only Strings. // Returns a list of appended strings.
List12 appendStrings (List12 strList) {
    List12 appendedStrList = new ArrayList();
    final Iterator iterator = strList.iterator();
    while (iterator.hasNext()) {
        appendedStrList.add("appendage" + (String) list.next());
    }
    return appendedStrList;
}
```
More plausible example

A more plausible example of the problem above might occur if you are implementing a method that takes a collection as a parameter:

```java
// Specified list should contain only Strings.
// Returns a list of appended strings.
List12 appendAndPrint (List12 strList) {
    List12 appendedStrList = new ArrayList();
    final Iterator iterator = list.iterator();
    while (iterator.hasNext()) {
        appendedStrList.add(
            "appendage" + (String) list.next()
        );
    }
    return appendedStrList;
}
```

If user passed in a list that contained any non-String object, then we’ll get a `ClassCastException`.
Naive fix

- How can we fix the problems of tedium, ugly code, and potential ClassCastException?

- One naive strategy is to define a different ArrayList for every class we want to store in it, e.g.:

```java
class ArrayListOfStrings {
    public void add (String s) { ... }
    public String get (int index) { ... }
}
class ArrayListOfIntegers {
    public void add (Integer i) { ... }
    public Integer get (int index) { ... }
}
class ArrayListOfShapes {
    public void add (Shape s) { ... }
    public Shape get (int index) { ... }
}
```

With specific types, we no longer have to downcast the result, and we're guaranteed that `get(index)` returns a String.
Naive fix

• However, this “naive fix” is very tedious -- we have to create another version of the `ArrayList` for every class we want to support.

• “Copying+pasting code” would save some time, but this is never a good idea.

• Inevitably, one of the `ArrayListOfX` classes will change, and you’ll forget to change the other ones correspondingly.

• Let’s take another look at those “related classes”...
Better fix: factor out the type

```java
class ArrayListOfStrings {
    public void add (String s) { ... }
    public String get (int index) { ... }
}
class ArrayListOfIntegers {
    public void add (Integer i) { ... }
    public Integer get (int index) { ... }
}
class ArrayListOfShapes {
    public void add (Shape s) { ... }
    public Shape get (int index) { ... }
}
```

- The only place these class definitions differ is in the type of the objects they hold.
- It seems like there should be a way to “factor out” the type...
Java generics.
Java generics

- Since Java 1.5, Java has offered the ability to parameterize a class by a type.
- For example, when writing a “collection” class such as ArrayList, we can give it a type parameter T.
- As with data parameters, the parameter name is up to the programmer.
- Type parameters are typically given one-letter names:
  - K for “key”, V for “value”, E for “element, etc.”
Generics for “ArrayListOfX”

- Consider our problem of writing multiple ArrayListOfX classes to store data of different types:

- With Java generics, we can write just one version of the class and parameterize it by type \( \text{T} \), the type of data the ArrayList should contain.
Generics for “ArrayListOfX”

class ArrayList<T> implements List<T> {
    T[] _underlyingStorage;
    int _numElements;

    void add (T element) {
        _underlyingStorage[_numElements] = element;
        _numElements++;
    }

    T get (int index) {
        return _underlyingStorage[index];
    }
}

The type parameter T is specified in angled brackets just after the classname. Thereafter, it can be used inside the class anywhere a type is expected. (Almost -- more later.)

Interfaces too can be parameterized by a type.
Generics for “List0fx”

• Similarly to classes, interfaces too can be parameterized by a type:

```java
interface List<T> {
    void add (T element);
    T get (int index);
    void remove (int index);
}
```
Generics for "ArrayListOfX"

• In short: (almost) everywhere in our previous versions of List and ArrayList, we replace the type Object with the type parameter T.

• To instantiate the “generic” ArrayList<T> in code:

  When we instantiate the generic collection, we must specify the value of the type parameter.

  ```java
  final ArrayList<Student> list = new ArrayList<Student>();
  ```

• Instantiating the ArrayList with type parameter T=Student can be conceptualized as doing a “search-and-replace” to change <T> to <Student>:

  ```java
class ArrayList<T> ... {
    void add (T element) { ... }
    T get (int index) { ... }
}

  ```

  ```java
class ArrayList ... {
    void add (Student element) { ... }
    Student get (int index) { ... }
}
  ```
Generics for “ArrayListOfX”

- Now, our list can only be populated with Student data (or any subclass of Student):

  ```java
  list.add(new Student());  // -- ok by definition
  list.add(new UCSDStudent());  // -- ok if it’s a subclass
  ```

- What happens if we try to break this rule and add a non-Student object to list?

  ```java
  list.add("error");  // not ok -- compiler catches this!
  ```
Generics for “ListOfX”

- Now, our list can only be populated with Student data (or any subclass of Student):
  
  ```java
  list.add(new Student());  // -- ok by definition
  list.add(new UCSDStudent());  // -- ok if it’s a subclass
  ```

- What happens if we try to break this rule and add a non-Student object to list?
  
  ```java
  list.add("error");  // not ok -- compiler catches this!
  ```

- With Java generics, the compiler will catch this error -- it knows that “error” is a String, and that list is of type `ArrayList<Student>`.

- Since `ArrayList<Student>`’s `add(element)` method expects a Student, there is a type mismatch -- compile-time error.
Benefits of generics

• It is preferable for the compiler to catch this mistake rather than the JVM:
  • We fix the bug before the program crashes.
  • The compiler rules out the possibility that we mismatch container type and element type.

• With generics, we also no longer have to downcast the return value of `get(index)`:

```java
final ArrayList<String> list = new ArrayList<String>();
list.add("hello");
final String s = list.get(0);  // No downcast necessary
```

• This is because the result of `get(index)` is guaranteed to be of type `String` -- we don’t have to additionally “promise” the compiler anything.