12
Abstract Data Types

Please note that these slides are based in part on material originally developed by Prof. Kevin Wayne of the CS Dept at Princeton University.

Abstract Data Types

- In the introduction, I referred to ADTs without definition. So…
  - *Data types*: A data type is a set of values and a set of operations on those values.
  - *Abstract data types*: An abstract data type is a data type whose internal representation is hidden from the client (encapsulation).
  - *Objects*: An object is an entity that can take on a data-type value. Objects are characterized by three essential properties: The state of an object is a value from its data type; the identity of an object distinguishes one object from another; the behavior of an object is the effect of data-type operations. In Java, a reference is a mechanism for accessing an object.

Application Programming Interface

- We briefly referred to an API earlier, without defining it. It stands for Application Programming Interface and forms the *contract* between the programmer of a class and its clients.
- To specify the behavior of an abstract data type, we use an application programming interface (API), which is a list of constructors, instance and class methods, and public fields (instance/class)—if any—with an informal description of the effect of each.
    - *Client*: A client is a program that uses a data type.
    - *Implementation*: An implementation is the code that implements the data type specified in an API.

ADT/API example

API

public class Counter		
	Counter(String id)	*create a counter named* id
void	increment()	*increment the counter by one*
int	tally()	*number of increments since creation*
String	toString()	*string representation*

- instantiation: `Counter heads = new Counter("heads");`
- increment: `heads.increment();`
- tally: `if (heads.tally() > 100) return;`
- show: `System.out.println("Heads: "+heads);`

Note that we don't know anything about the internal details. The client code is just following the API

Counter

```java
package edu.neu.coe.info6205;
public class Counter {
    public Counter(String id) {
        this.id = id;
    }
    public void increment() {
        count++;
    }
    public int tally() {
        return count;
    }
    @Override
    public String toString() {
        return id+": "+count;
    }
    private final String id;
    private int count = 0;
}
```

Counter--usage

```
package edu.neu.coe.info6205;
import java.util.Random;
public class Counter {
    ...
    public static void main(String[] args) {
      Counter heads = new Counter("heads");
      Counter tails = new Counter("tails");
      Random random = new Random();
      for (int i = 0; i < 100; i++) if (random.nextBoolean()) heads.increment(); else
tails.increment();
      System.out.println(heads);
      System.out.println(tails);
    }
    ...
}
```

Implementing ADTs

- *Encapsulation.*
  - A hallmark of object-oriented programming is that it enables us to *encapsulate* data types within their implementations, to facilitate separate development of clients and data type implementations. Encapsulation enables modular programming.
  - Also known as *information-hiding*.

Implementing ADTs (continued)

- *Designing APIs.*
  - One of the most important and challenging steps in building modern software is designing APIs. Ideally, an API would clearly articulate behavior for all possible inputs, including side effects (if any), and then we would have software to check that implementations meet the specification. Unfortunately, a fundamental result from theoretical computer science known as the *specification problem* implies that this goal is actually impossible to achieve. There are numerous potential pitfalls when designing an API:
    - Too hard to implement, making it difficult or impossible to develop.
    - Too hard to use, leading to complicated client code.
    - Too narrow, omitting methods that clients need.
    - Too wide, including a large number of methods not needed by any client.
    - Too general, providing no useful abstractions.
    - Too specific, providing an abstraction so diffuse as to be useless.
    - Too dependent on a particular representation, therefore not freeing client code from the details of the representation.

Implementing ADTs (continued)

- ★Brevity is the soul of wit (William Shakespeare, *Hamlet*)
- ★In summary, provide to clients the methods they need and no others.

Implementing ADTs (contd.)

- *Algorithms and ADTs.* Since data structures and algorithms go together, data *abstraction* is naturally suited to the study of algorithms, because it helps us provide a framework within which we can precisely specify both what an algorithm needs to accomplish and how a client can make use of an algorithm. For example, our whitelisting example from the previous lecture is naturally cast as an ADT client, based on the following operations:
    - Construct a SET from an array of given values.
    - Determine whether a given value is in the set.

Implementing ADTs (contd.)

- *Interface inheritance.* Java provides language support for defining relationships among objects, known as *inheritance*. Interfaces are the answer to too much *coupling*. We use interface inheritance for *comparison* and for *iteration*.
- *Inheritance via sub-classing.* Any (non-final) class can be sub-classed. However, people are not that good at recognizing such situations. In Java, every class is a sub-class of Object which defines several important methods, including *toString, equals and hashCode*.

Sidetrack: Coupling

- You likely won't know what coupling is unless you've worked on a big project.
  - Well-architected software systems have low coupling (and high cohesion—where related concepts are in the same module);
  - If every little change you make to a software module ripples through many other modules: that's tightly coupled (bad).
  - Suppose ADT *X* has the following signature:
    - *public Y getY()*
  - And suppose ADT *Y* has the following signature:
    - *public X getX()*
  - You've got some tight coupling. If you want to change a different signature in *X*, you probably will end up also having to change *Y* too.
- Encapsulation also helps to avoid coupling by hiding implementation details.
- Interfaces are the chief way to reduce coupling. Use them!

Interfaces

	interface	methods	section
comparison	`java.lang.Comparable`	`compareTo()`	2.1
	`java.util.Comparator`	`compare()`	2.5
iteration	`java.lang.Iterable`	`iterator()`	1.3

	`java.util.Iterator`	`hasNext()`
		`next()`
		Ignore: `remove()`
			1.3

Interfaces: iteration

```
public interface Iterable<T> {
    /**
     * Returns an iterator over elements of type {@code T}.
     * @return an {@code Iterator<T>}.
     */
    Iterator<T> iterator();
}

public interface Iterator<T> {
    /**
     * Method to determine if this iterator has more elements.
     * This allows the caller to know if a call to {@code next()} would
     * return a {@code T} as opposed to throwing an exception.
     * @return {@code true} if the iteration has more elements
     */
    boolean hasNext();

    /**
     * @return the next element in the iteration
     * @throws {@code NoSuchElementException} if the iteration has no more
elements
     */
     T next();
}
```

Iteration in practice

```
Bag<Integer> bag = new Bag_Array<>();
for (int i = 1; i <= 4; i++)
    bag.add(i);
int sum = 0;
for (Integer x : bag) sum += x;
assertEquals(10, sum);
```

Interfaces, detail: *Comparable*

```
public interface Comparable<T> {
    /**
     * Compares this object with the specified object for order.  Returns a negative
integer, zero, or a positive integer as
     * this object is less than, equal to, or greater than the specified object.
     *
     * <p>The implementor must ensure <tt>sgn(x.compareTo(y)) == -
sgn(y.compareTo(x))</tt> for all <tt>x</tt> and <tt>y</tt>.  (This implies that
<tt>x.compareTo(y)</tt> must throw an exception iff <tt>y.compareTo(x)</tt> throws
an exception.)
     *
     * <p>The implementor must also ensure that the relation is transitive:
<tt>(x.compareTo(y)&gt;0 && y.compareTo(z)&gt;0)</tt> implies
<tt>x.compareTo(z)&gt;0</tt>.
     *
     * <p>Finally, the implementor must ensure that <tt>x.compareTo(y)==0</tt>
implies that <tt>sgn(x.compareTo(z)) == sgn(y.compareTo(z))</tt>, for all
<tt>z</tt>.
     *
     * @param   o the object to be compared.
     * @return  a negative integer, zero, or a positive integer as this object is
less than, equal to, or greater than the specified object.
     * @throws NullPointerException if the specified object is null
     * @throws ClassCastException if the specified object's type prevents it from
being compared to this object.
     */
    public int compareTo(T o);
}
```

Interfaces, detail: *Comparator*

```
public interface Comparator<T> {
    /**
     * Compares its two arguments for order.  Returns a negative
integer, zero, or a positive integer as the first argument is
less than, equal to, or greater than the second.<p>
     *
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, zero, or a positive integer as
the first argument is less than, equal to, or greater than the
second.
     * @throws NullPointerException if an argument is null and
this comparator does not permit null arguments
     * @throws ClassCastException if the arguments' types prevent
them from being compared by this comparator.
     */
    int compare(T o1, T o2);
}
```

Sub-classing Object

	method	purpose	section
Class	getClass()	*what class is this object?*	1.2
String	toString()	*string representation of this object*	1.1
boolean	equals(Object that)	*is this object equal to* that*?*	1.2
int	hashCode()	*hash code for this object*	3.4

- Please note that these methods aren't just for fun. They are really important (as you'd expect for methods that are defined for ***every*** object.
- One criticism I have of Java (there are others in my "Software" blog) is that they should have defined ***equals*** and ***hashCode*** as part of an interface whereby both (or neither) methods should be defined. That's because it can be really problematic if these two aren't compatible!

More on Equality

Equality. What does it mean for two objects to be equal? If we test equality with `(a == b)` where `a` and `b` are reference variables of the same type, we are testing whether they have the same identity: whether the ***references*** are equal. We also need a way to test logical or datatype equality. This is defined in the method ***equals()***. When we define our own data types we need to override `equals()`. Java's convention is that `equals()` must be an ***equivalence relation***:

- *Reflexive:* `x.equals(x)` is true.
- *Symmetric*: `x.equals(y)` is true if and only if `y.equals(x)` is true.
- *Transitive*: if `x.equals(y)` and `y.equals(z)` are true, then so is `x.equals(z)`.

In addition, it must take an `Object` as argument and satisfy the following properties.

- *Consistent*: multiple invocations of `x.equals(y)` consistently return the same value, provided neither object is modified.
- *Not null*: `x.equals(null)` returns false.

More about Java: the practicalities

- *Memory management.* One of Java's most significant features is its ability to ***automatically*** manage memory. When an object can no longer be referenced, it is said to be ***orphaned***. Java keeps track of orphaned objects and returning the memory they use to a pool of free memory. Reclaiming memory in this way is known as ***garbage collection***.
- *Immutability.* An ***immutable*** data type has the property that the value of an object never changes once constructed. By contrast, a ***mutable*** data type manipulates object values that are intended to change. Java's language support for helping to enforce immutability is the `final` modifier. When you declare a variable to be `final`, you are promising to assign it a value only once, either in an initializer or in the constructor. Code that could modify the value of a `final` variable leads to a compile-time error. Vector.java is an immutable data type for vectors. In order to guarantee immutability, it ***defensively copies*** the mutable constructor argument.

More about Java: the practicalities (2)

- *Exceptions and errors* are disruptive events that handle unforeseen errors *outside* our control. We have already encountered the following exceptions and errors:
  - *ArithmeticException*. Thrown when an exceptional arithmetic condition (such as integer division by zero) occurs.
  - *ArrayIndexOutOfBoundsException*. Thrown when an array is accessed with an illegal index.
  - *NullPointerException*. Thrown when `null` is used where an object is required.
  - ClassCastException. Thrown we we try to cast an object to a class which is non-assignable.
  - *OutOfMemoryError*. Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory.
  - *StackOverflowError*. Thrown when a recursive method recurs too deeply.
- You can also create your own exceptions. The simplest kind is a *RuntimeException* that terminates execution of the program and prints an error message.
  - throw new RuntimeException("Error message here.");

More about Java: the practicalities (3)

- *Assertions* are boolean expressions which verify assumptions that we make ***within*** code we develop. We can use this mechanism to test assumptions and/or invariants. If the expression is false, the program will terminate and report an error message. For example, suppose that you have a computed value that you might use to index into an array. If this value were negative (or too large), it would cause an `ArrayIndexOutOfBoundsException` sometime later. But if you write the code
    - `assert i >= 0 && i < N, “index out of bounds”;`
- You can pinpoint the place where the error occurred. By default, assertions are disabled.You can enable them from the command line by using the `-enableassertions` flag (`-ea` for short). Assertions are for debugging: your program should not rely on assertions for normal operation since they may be disabled.

Mystery recursive function

```
public static String mystery(String s) {
    int N = s.length();
    if (N <= 1) return s;
    String a = s.substring(0, N/2);
    String b = s.substring(N/2, N);
    return mystery(b) + mystery(a);
}
```