ADT Introduction

Part 1 of 2
What is a data type?

A data type is composed of two sets
- a set of data values
- a set of operations on the data

In OOP a data type corresponds to a class

Each class defines
- a set of objects (the set of data values)
- a set of methods that can be applied to an object of the class

ADT are implemented using data structures
What is an ADT?

- ADT = abstract data type
- The word abstract means that we are considering only the specification of the data type not its implementation
- Often an ADT is specified in an abstract sense without reference to a particular programming language
- Sometimes ADT and DT are synonymous
Built-in data types

- Integer numeric types
  - byte, char, short, int, long
- Floating point numeric types
  - float, double
- Other primitive types
  - boolean
- Array type
  - arrays of primitive or reference types
Data for primitive types

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Minimum value</th>
<th>Maximum value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2147483648 = $-2^{31}$</td>
<td>2147483647 = $2^{31} - 1$</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>$\pm 1.40 \times 10^{-45}$</td>
<td>$\pm 3.40 \times 10^{38}$</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>$\pm 4.94 \times 10^{-324}$</td>
<td>$\pm 1.80 \times 10^{308}$</td>
</tr>
</tbody>
</table>

There is also the boolean type with the two values true or false
Operations for primitive types

- **byte, char, short, int, long types**
  - +, -, *, /, %, convert to string, etc

- **float, double types**
  - +, -, *, /, round, ceil, floor, etc.

- **boolean type**
  - test for true, test for false

See java.lang.Math
String type

- This is an object type that is fundamental to almost all Java programs
- data
  - strings with 0 or more characters
- operations
  - many operations in the `String` class such as
    - `+` (concatenation), `substring`, `charAt`,
- Strings are immutable (no `setCharAt`)
StringBuilder type (Java 5)

- Replaces `StringBuffer` for non-threaded applications

- **Data**
  - A mutable dynamic version of the `String` type (size grows as needed).

- **Operations**
  - `append`, `charAt`, `setCharAt`, `toString`, etc
Example: The Bank ADT

Data:
- Bank objects containing accounts

Operations:
- size: return number of accounts in bank
- add: add new account to end of bank
- get: return account at a given index (0,1,...)
- set: replace the account at given index
- clone: return a copy of a bank
- toString: return string representation of bank
Example: The Motor ADT

Data:
- Motor objects that have a speed lever (0 to 3600 rpm), a direction lever (forward, reverse), and an on/off button

Operations:
- getDirection, setDirection (forward or reverse)
- getSpeed, setSpeed (0 to 3600 rpm)
- getOn (true/false), setOn
- getOff (true/false), setOff
Example: The CardDeck ADT

- **Data:**
  - A CardDeck contains 52 Card Objects
  - A Card has a rank and a suit and operations to get rank, suit and card names.

- **CardDeck Operations:**
  - **shuffle**: shuffle a deck of cards
  - **deal**: deal a card from the deck
  - **cardsInDeck**: return number of cards in deck
  - **empty**: test if a card deck is empty
Interfaces and ADTs

- A Java interface can be used to specify an ADT.
  - We could have done this with our simple Bank Motor, or CardDeck examples
- The interface contains method prototypes that specify the ADT operations but no implementations
Collections

- A common kind of ADT is called a collection.
- A collection (collection class in Java) is a grouping together of related objects
- Example: the bank accounts in a bank
- Example: the cards in a deck of cards
- A simple way to implement a collection class is to use an array as a data field
- We did this with a simple Bank class
Important collections (1)

- Set data type
  - no duplicates and the order is unimportant

- Bag data type
  - (multi-set): duplicates are allowed

In both cases the order is unimportant
Important collections (2)

- **Sequence data type**
  - relative/indexed access

- **List data type**
  - relative/indexed access

- **Stack data type**
  - size, push, pop, top (peek)

- **Queue data type**
  - size, enqueue, dequeue, first, last
What is a data structure?

- A data structure is a programming construct used to store and organize the data in an implementation of an ADT.
- A specification of an ADT normally does not mention the data structures used: that is an implementation detail.
- An ADT can have several implementations each differing in the data structure chosen.
- Some structures are more efficient than others.
Kinds of data structures

- Linear data structures (COSC 2006)
  - Fixed size array
  - Dynamic array
  - Singly linked list
  - Doubly linked list

- Non-linear data structures (COSC 2007)
  - trees
  - graphs
  - hash tables
Direct/relative access

- Two ways to provide access to an ADT
  - direct access using an index (absolute access)
  - relative access (before, after)
- An array structure provides efficient $O(1)$ direct access using an index but insertions and deletions are $O(n)$ in general
- A list can provide $O(1)$ insertions and deletions but is inefficient for indexing
Fixed size array

- This is the only data structure that we discussed in first year.
- An array can be used to store data of a primitive type or references to data of object type.
- Once constructed size cannot be changed

```
```

8/23/2007
Partial fixed array

- A collection may not use the entire array.
- Initially the collection is empty.
- When the array is full no more elements can be added to the collection.

In Use (size is 3)  
Unused Part
Dynamic array

- Begin with a fixed size array
- To add an element to a full array the following steps are performed
  - make a double size copy of the fixed array
  - copy the elements currently in the full array to the new array
  - Make the original array reference refer to the new array
  - original array can now be garbage collected
Dynamic array (1a)

First make an array referenced by copy that is twice the size of the array referenced by a
Dynamic array (1b)

Now copy the data in original array to the new one
Now change the reference "a" to reference the copy. The original array will be garbage collected.
When the method exits the local reference copy disappears and the original array will eventually be deleted by the garbage collector.
Singly linked list

- A link in an object is a variable containing a reference to another object of same type

A node

- A singly linked list is some linked nodes
Comparison

- Array

- Linked List
Doubly linked list

The nodes of a doubly linked list have references in both directions.
COSC 2006: Data Structures I

ADT Introduction
Part 2 of 2
We can double the size of an array of type BankAccount using the following method. For example if we have an array called bank the following method doubles the size of this array

```java
private void reallocate()
{
    int newCapacity = 2 * bank.length;
    BankAccount[] newBank =
        new BankAccount[newCapacity];
    for (int k = 0; k < bank.length; k++)
    {
        newBank[k] = bank[k];
    }
    bank = newBank;
}
```
Using System.arraycopy

- System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

  - This method is more efficient than for loop
  - src is the source array and srcPos is the starting position in this array
  - dest is the destination array and destPos is the position to start copying to
  - length is the number of elements to copy
The following version of the `reallocate()` method uses the `arraycopy` method instead of an explicit for loop.

```java
private void reallocate()
{
    int newCapacity = 2 * bank.length;
    BankAccount[] newBank =
        new BankAccount[newCapacity];
    System.arraycopy(bank,0,newBank,0,
                     bank.length);
    bank = newBank;
}
```
This version of the Bank ADT uses a dynamic array. When the array is filled it is doubled in size

```java
package bankexamples.dynamic;
import bankexamples.accounts.BankAccount;
public class Bank implements Cloneable {
    private BankAccount[] bank; // array adapter
    private int size; // number of accounts
```
Default constructor for a Bank of 10 accounts and general constructor for a given initial capacity. When this capacity would be exceeded the bank is doubled in size.

```java
public Bank()
{  this(10);
}
public Bank(int initialCapacity)
{
    if (initialCapacity < 1)
    {  throw new IllegalArgumentException(
            "Initial size must be positive");
    }
    bank = new BankAccount[initialCapacity];
    size = 0;
}
```
Reallocate space: This method is called when the add method is called and the bank array is full. The method doubles the size of the array and copies existing element to the new array.

```java
private void reallocate() {
    int newCapacity = 2 * bank.length;
    BankAccount[] newBank =
        new BankAccount[newCapacity];

    for (int k = 0; k < bank.length; k++) {
        newBank[k] = bank[k];
    }

    bank = newBank;
}
```
Dynamic Bank (4)

Return the current number of accounts in the bank. We always have $0 \leq size < bank.length$

```java
public int size()
{
    return size;
}
```
Add a new account at the end of the bank array if there is room. Otherwise call reallocate to double the size of the array. In any case size gives the position of the element added.

```java
public void add(BankAccount b) {
    if (size() == bank.length) {
        reallocate();
    }
    bank[size] = b;
    size++;
}
```
Return a reference to the account with a given index. If the index is out of range then throw an exception

```java
public BankAccount get(int index)
{
    if (index < 0 || index >= size())
    {
        throw new IllegalArgumentException("invalid index for get");
    }
    return bank[index];
}
```
Replace the account at a given index by a new one and return a reference to the account that was replaced. If the index is out of range an exception is thrown.

```java
public BankAccount set(int index, BankAccount b) {
    if (index < 0 || index >= size()) {
        throw new IllegalArgumentException("invalid index for set");
    }
    BankAccount original = bank[index];
    bank[index] = b;
    return original;
}
```
Dynamic Bank (8)

Make a shallow clone of a Bank object. `super.clone()` is called to make a copy of the instance data fields. Then the array of references is cloned using the array clone.

```java
public Bank clone() {
    Bank copy;
    try {
        copy = (Bank) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException(
            "Cannot clone bank");
    }
    copy.bank = bank.clone();
    return copy;
}
```

Java 5
Create a string representation of a bank by using the toString method in the bank class

```java
public String toString()
{
    if (size() == 0) return "Empty list";
    StringBuilder s = new StringBuilder();
    for (int k = 0; k < size(); k++)
    {
        s.append(bank[k]);
        s.append("\n");
    }
    return s.toString();
}
} // end of Bank class
```
Test the dynamic array:
Create a bank that can hold one account.
Add three accounts to it and display the bank using toString

See class BankTester in package bankexamples.dynamic

```java
Bank bank = new Bank(1);
bank.add(new BankAccount(123, "Fred", 100.0));
bank.add(new BankAccount(124, "Mary", 150.0));
bank.add(new BankAccount(125, "Gord", 200.0));
System.out.println(bank);
```
Use get to display the accounts

```
for (int k = 0; k < bank.size(); k++)
{
    System.out.println(bank.get(k));
}
System.out.println();
```
Add $100 to all accounts. This is done by getting a reference to each account and using the deposit method to change the balance. This works because the BankAccount class is mutable.

```
for (int k = 0; k < bank.size(); k++)
{
    BankAccount b = bank.get(k);
    b.deposit(100);
}
System.out.println(bank);
```
Change the account numbers to 1, 2, 3, ... by creating a new BankAccount and using the set method to replace the account at the given position with the new one.

```java
for (int k = 0; k < bank.size(); k++)
{
    BankAccount b = bank.get(k);
    BankAccount a = new BankAccount(k+1,
        b.getName(), b.getBalance());
    bank.set(k, a);
}
System.out.println(bank);
```
Clone the bank and change the balance of the account at index 0. This change is seen by bank and its clone bankClone since they are sharing the objects.

```java
Bank bankClone = (Bank) bank.clone();
BankAccount b = bank.get(0);
b.deposit(1000);
System.out.println(bank);
System.out.println(bankClone);
```
Now create a new account and use the bank set method to replace the account at index 0 by this account. Now the bank object reflects this change but the clone doesn't since it still references the original account at index 0.

```java
BankAccount newAccount = new BankAccount(123, "NoName", 0.0);
bank.set(0, newAccount);
System.out.println(bank);
System.out.println(bankClone); // original
```
Program to an interface

- Program to an interface not an implementation
- Design as much as you can with interfaces
- Write as much code as you can using only the interfaces
- Then make classes that implement the interfaces.
- Several different classes can implement the same interface.
Programming to an interface (1)

- We have written two versions of the simple Bank class
  - `bankexamples.simple.Bank`: version that uses a fixed array for the implementation
  - `bankexamples.dynamic.Bank`: version that uses a dynamic array for the implementation
- Both versions have the same methods
- We can express this better by using a SimpleBank interface.
Programming to an interface (2)

interface SimpleBank

- methods: size, add, get, set, toString

Write two implementations of this interface

- bankexamples.implementors.FBank: a fixed array implementation
- bankexamples.implementors.DBank: a dynamic array implementation

Can include other methods not in interface

- Example: clone
There may be several ways to implement a Bank so we provide an interface.

```java
package bankexamples.interfaces;
import bankexamples.accounts.BankAccount;
public interface SimpleBank {
    public int size();
    public void add(BankAccount b);
    public BankAccount get(int index);
    public BankAccount set(int index, BankAccount b);
    public String toString();
}
```
This interface specifies 5 methods: 
size, add, get, set, toString

package bankexamples.interfaces;
import bankexamples.accounts.BankAccount

class SimpleBank
{
/**
 * Return number of accounts currently in bank.
 * @return number of accounts currently in bank.
 */

public int size();
/**
 * Add another account to the bank after the last one.
 * @param b the account to add
 */

public void add(BankAccount b);
/**
 * Return account at given index.
 * @param index the account index
 * @return account at given index
 * @throws IllegalArgumentException
 *     if index < 0 or index >= size()
 */

public BankAccount get(int index);
/**
 * Replace object at position index with a new one and return the original object.
 * @param index the index of the account
 * @param b the new bank account.
 * @throws IllegalArgumentException if index < 0 or index >= size()
 */

public BankAccount set(int index, BankAccount b);
/**
 * Return string representation of the bank.
 * @return string representation of the bank.
 */

public String toString();

} // end of interface SimpleBank
SimpleBank Documentation

- It is important that the complete javadoc description be given for the interface and its methods.
- This documentation does not need to be repeated in each class that implements the interface.
- The interface and its javadoc provides the specification of the SimpleBank ADT
Implementing SimpleBank

The Bank class discussed earlier can be made to implement this interface: write implementations of each interface method and provide the constructors and any other useful methods.

```java
package bankexamples.implementors;
import bankexamples.accounts.BankAccount;
import bankexamples.interfaces.SimpleBank;
public FBank implements SimpleBank
{
    public FBank() {...}
    public FBank(int maxSize) {...}

    public int size() {...}
    public void add(BankAccount b) {...}
    public BankAccount get(int index) {...}
    public BankAccount set(int index, BankAccount b) {...}
    public Object clone() {...}
    public String toString() {...}
}
```
Since FBank implements SimpleBank we can write

```
SimpleBank bank = new FBank(50);
```

instead of

```
FBank bank = new FBank(50);
```
FBank implementation (1)

A version of the fixed size array implementation of the Bank class that shows how a class can implement interfaces

```java
package bankexamples.implementors;
import bankexamples.interfaces.SimpleBank;

public class FBank implements SimpleBank, Cloneable {
    private BankAccount[] bank;
    private int size;
    private int maxSize;
}
```
Constructors are never part of the the interface so we always need to implement them. Different implementations may choose different constructors.

```java
public FBank()
{   this(10);
}

public FBank(int maxSize)
{
    bank = new BankAccount[maxSize];
    size = 0;
    this.maxSize = maxSize;
}
```
Instead of writing the javadoc comments again we can just refer to the comments in the interface. Eclipse can produce these references automatically.

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#size()
 */

public int size()
{
    return size;
}
```
We have done the add method before:

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#add(bankexamples.accounts.BankAccount)
 */

public void add(BankAccount b)
{
    // done before
}
```
FBank implementation (5)

We have done the get method before:

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#get
 */

public BankAccount get(int index)
{
    // done before
}
```
We have done the set method before:

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#set(int,bankexamples.accounts.BankAccount)
 */

public BankAccount set(int index,
                        BankAccount b)
{
    // done before
}
```
A stub was not generated for the toString method since any class automatically has a toString method from the Object class. Here we override it

```java
/**
 * Return a string representation of the bank.
 * @return string representation of the bank.
 */
public String toString()
{
    // done before
}
```
Override the clone method.
This is an extra method here that is not required to implement the SimpleBank interface.

```java
/**
 * Shallow clone of this bank.
 * @return shallow clone of this bank.
 */

public FBank clone()
{
    // done before
}

} // end of FBank class
```
Dynamic implementation

We could also do a dynamic implementation of the SimpleBank interface that grows the array when its maximum size is reached. (dynamic array implementations to be discussed later)

```java
package bankexamples.implementors;
import bankexamples.accounts.BankAccount;
import bankexamples.interfaces.SimpleBank;
public DBank implements SimpleBank {
    public DBank() {...}

    // initialCapacity is the initial array size
    public DBank(int initialCapacity) {...}

    // include the methods here that implement
    // the SimpleBank interface and any other
    // methods that are useful.
}
```
A version of the dynamic array implementation of the Bank class that shows how a class can implement an interface

```java
package bankexamples.implementors;
import bankexamples.interfaces.SimpleBank;

public class DBank implements SimpleBank {
    private BankAccount[] bank;
    private int int size;
}
```
Constructors are never part of the interface so we always need to implement them. Different implementations may choose different constructors.

```java
public DBank()
{
    this(10);
}

public DBank(int initialCapacity)
{
    // done before
}
```
Instead of writing the javadoc comments again we can just refer to the comments in the interface. Eclipse can produce theses references automatically.

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#size()
 */

public int size()
{
    return size;
}
```
We have done the add method before:

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#add(bankexamples.accounts.BankAccount)
 */

public void add(BankAccount b) {
    // done before
}
```
We have done the get method before:

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#
 * get(int)
 */

public BankAccount get(int index)
{
    // done before
}
```
We have done the set method before:

```java
/* (non-Javadoc)
 * @see bankexamples.interfaces.SimpleBank#set(int,bankexamples.accounts.BankAccount)
 */

public BankAccount set(int index, BankAccount b) {
    // done before
}
```
A stub was not generated for the toString method since any class automatically has a toString method from the Object class. Here we override it.

```java
/**
 * Return a string representation of the bank.
 * @return string representation of the bank.
 */
public String toString()
{
    // done before
}
```
Override the clone method
This is an extra method here that is not required to implement the SimpleBank interface.

```
/**
 * Shallow clone of this bank.
 * @return shallow clone of this bank.
 */

public DBank clone()
{
    // done before
}

} // end of DBank class
We haven't discussed linked implementations of collection classes yet but a linked implementation is automatically dynamic so we only need the default constructor.

```java
package ...;
import ...
public LinkedBank implements SimpleBank {
    public LinkedBank() {...}
    // include the methods here that implement
    // the SimpleBank interface and any other
    // methods that are useful.
}
```
Use box like this for comment

Put code here