Unit 3 – Abstract Data Types

Unit 3.1 – The Abstract Data Type (ADT)

We will use the abbreviation **ADT** often in this class. It is synonymous with **data structure**. The name is important because it tells us something about it. So let's look at the words separately.

A **data type** is just what it sounds like. You are familiar with int, float, etc. Those are types of data that have their own rules and limitations. For example there's a largest int and a smallest int. If you get into the binary you know that negative numbers are stored in two's complement form.

Java, like other programming languages, allows us to define new data types. An array is a data structure and data type. Arrays have the limitation that all the elements are of the same type (homogeneous, although see the section on polymorphism later in this unit for more about this). We can also define data types that have mixes of different types of data (heterogeneous).

The evolution of data types is interesting. In C language you can use the keyword **struct** to define types that have mixed types. For example you can define a struct that contains an int, a float and a char array inside the same struct. Here's an example:

```
struct MyStruct
{
  int size;
  float offset;
  char name[50];
};
```

When you are ready to create one of these structs you do so like this:

```
struct MyStruct myStruct;
```

and then you access the the fields inside the struct with the **field selection operator** "." like this:

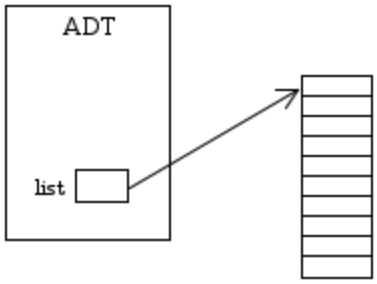
```
myStruct.size=100;
myStruct.offset=.5f;
strcpy(myStruct.name,"Whatever");
```

Note that Java still uses the same field selection operator today. In fact the modern Java class descended from the original C language struct with the addition of methods to operate on the fields, constructors, etc (this actually happened in C++).

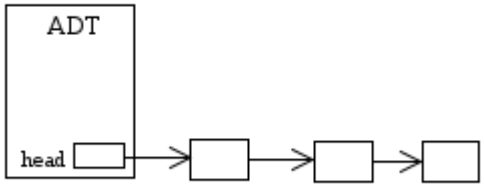
The next part of the name is the word **abstract**. Our ADTs are abstract because **the operations on the data within the ADT are specified independent of the implementation**. Let's look at a simple

example. When we study lists we will create two implementations: an array-based list and a linked list.

The array-based list can be sketched like this:



The linked list can be sketched like this:



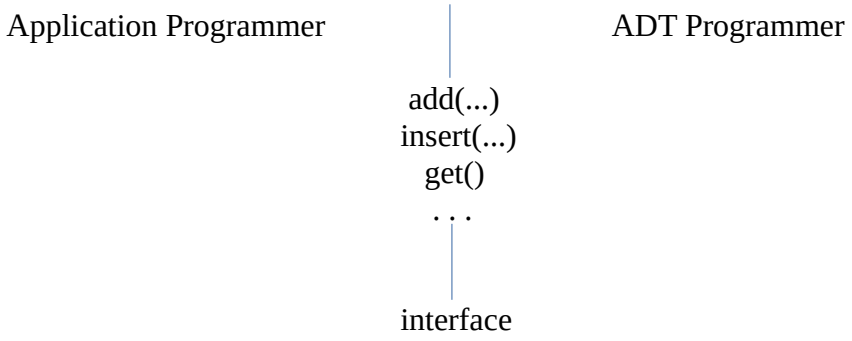
Both ADTs satisfy the List specifications but they way they work internally is completely different. Their behavior is different as well, for example in the array you can go directly to any element whereas in the linked list you need to traverse the list to get to an arbitrary element. The array-based list requires resizing when it runs out of room, whereas the linked list doesn't.

It's worth repeating that the List specification is called abstract because it is a set of fields and operations that don't rely on it's specific implementation.

Unit 3.2 – Interfaces

An interface is like a class but it has no executable code. Think of it as a list of method signatures. It is used in inheritance in a way that is similar to class inheritance. The reason we have interfaces is to maintain consistency between programmers that are working on different projects or different parts of the same project. If the method signatures are specified ahead of time it helps different modules work together once they are developed. Here's a sketch that might help make the point. On the left is the application programmer, on the right is the ADT programmer.

Each programmer can work independently and be assured that they are using the methods the same way. The vertical line represents a boundary, or interface between the two activities.



Here's what the looks like in code for the List ADT. Both programmers start with a List interface (don't worry about the <E> now, we'll get into that later).

```
public interface CM307List<E>
{
  public void add(E e);
  public boolean insert(int n,E e);
  public E get(int n);
  public E remove(int n);
  public boolean isEmpty();
  public int size();
}
```

Note that the methods don't have bodies ({...} blocks). They are followed by semicolons.

The application programmer knows that they can create a list and call the various methods even if the ADT development is not complete. Of course they won't be able to test yet, so their development will be limited.

The ADT programmer can then proceed with an implementation based on this interface. For the array-based list that would look something like this:

```
public class CM307ArrayList<E> implements CM307List<E>
{
  ...
}
```

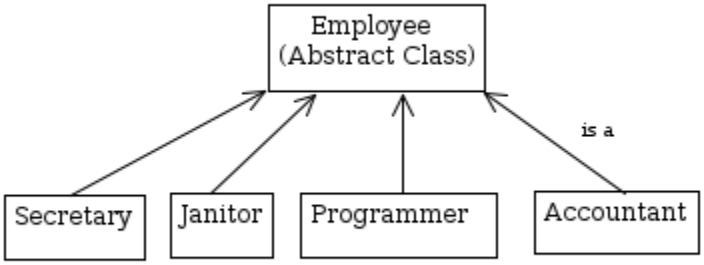
In order to compile the class CM307ArrayList must provide executable code for every method contained in the interface.

Unit 3.3 – Abstract Classes

No doubt you know about inheritance in Java classes. We can create classes (called subclasses) that extend other classes thereby inheriting all the fields and methods of the class we're extending (called the superclass). There are some interesting subtleties that are worth pointing out in the form of an example.

An **abstract class** is a class that is declared as abstract in its definition. Abstract classes cannot be instantiated. Classes that can be instantiated are known as **concrete classes**. We use abstract classes only to provide superclasses for inheritance, where the class is too general to be useful in describing any objects. One or more of the methods of the abstract class are declared as **abstract methods**. Abstract methods don't have a body, only a signature followed by a semicolon, similar to what we see in interfaces.

Imagine the inheritance tree below representing employees at an organization.



Unlike an interface the keyword abstract is required to mark a method as abstract. You can provide concrete methods in an abstract class. Just don't use the abstract keyword in the method definition.

Here's what the abstract class Employee might look like:

```
public abstract class Employee
{
  public Employee()
  {
    ...
  }

  public abstract void applyRaise(float pct);
  ...
}
```

So which methods should be abstract and which concrete? That depends on whether the same code can be applied to all subclasses. If it can, then the method should be concrete and is then inherited by all the subclasses. If the code is different in each subclass, then the method in the superclass should be abstract, forcing all the subclasses to provide concrete methods with executable code. The rule here is the same as with interfaces. If the subclass doesn't provide concrete methods for every abstract method

in the superclass, it won't compile, unless it too is marked as abstract.

The method applyRaise was chosen for this example since the various types of employees differ in how raises are calculated. An hourly employee such as a Secretary or Janitor will have their raises calculated differently from salaried or contract employees. Thus they need different code.

The subclasses may be defined like this:

```
public class Programmer extends Employee
{
  public Programmer()
  {
    super();
    ...
  }

  public void applyRaise(float pct)
  {
    ...
  }
}
```

The call to "super" in the Programmer class calls the superclass constructor. This isn't always necessary, it depends on what the Employee constructor does. If it does some important initialization, then it should be called.

If applyRaise(...) is the only abstract method, then the Programmer class is now a concrete class and can be instantiated.

There's an important point to make here regarding the difference between an abstract class and an interface. **A subclass can only extend one class but can implement an unlimited number of interfaces**. C++ allowed for multiple inheritance (extending more than one class) but the creators of Java decided not to support that.

Unit 3.4 – Using Interface and Abstract Class Names as Types

In the Employee inheritance hierarchy it should be clear that a Programmer "is a" Employee since it contains everything that any Employee object has in it (and more, since you can add new fields and methods in the Programmer class). So Java allows you to use a superclass reference variable to point to a subclass object. Here's an example:

```
Employee e=new Programmer();
```

However, there are some rules. When using the reference e, you can only call methods that are defined in the Employee class and inherited by the Programmer class. If you define new things in the Programmer class that are not inherited, it's like they don't exist. For example, let's say I introduce a

method in the Programmer class named setNerdLevel(int n) then the following won't compile:

```
e.setNerdLevel(10);
```

This may seem wrong, since the object pointed to by e is a Programmer object and has that method in it. The way to think about it is this: **the superclass Employee reference variable loses the Programmerness of the subclass**.

There is a way to get the Programmerness back, and that's with a cast. There's several ways to do that. here's one:

```
Programmer p=(Programmer)e;
p.setNerdLevel(10);
```

Or you can cast on the fly like this:

```
((Programmer)e).setNerdLevel(10);
```

Note that in this case extra parentheses are required to force the cast. We say that the field selection operator "." binds tighter than the cast. This is an operator precedence issue.

Note that no cast was required to assign a Programmer object into an Employee reference variable. That's because a Programmer "is a" Employee. A cast is required going the other way because not all Employee objects are Programmer objects.

The cast is like telling the compiler "trust me I know what I'm doing and the object pointed to by e will turn out to be a Programmer object". But what if it isn't? If you cast to Programmer and e is pointing to some other type of object the JVM (Java virtual machine) will throw a **ClassCastException** and your program will shut down.

There's another way to handle this situation. That is to use the **instanceof** operator. The syntax is simple:

```
if(e instanceof Programmer)
{
  ...
}
```

Unit 3.5 – Polymorphism

Polymorphism literally means "many forms". It means that an object can have many types at the same time. For example a Programmer object is also an Employee object. Trivially, it is also an Object object. Thus the many forms.

It also means that when you call a method defined in the superclass (and overridden in subclasses), on a subclass object, the JVM will redirect that call to the method defined in the appropriate subclass depending on what subclass the object is created from. This is called **dynamic binding**. The actual method to be called is selected at runtime. Here's an example.

Suppose you have an abstract method called applyRaise defined in the Employee class that is overridden with different code in each subclass. When you make the following call to applyRaise it will execute the code in the applyRaise method in the Programmer class:

```
Employee e=new Programmer();
e.applyRaise(.05f); //The "f" makes the number a float, .05 is a double
```

So the JVM actually checks what type of object e points to and executes the code in that object.

This is actually much more powerful than it may seem at first. Imagine a program that uses an array of type Employee to iterate through a set of subclass objects in order to apply raises. The code may look like this:

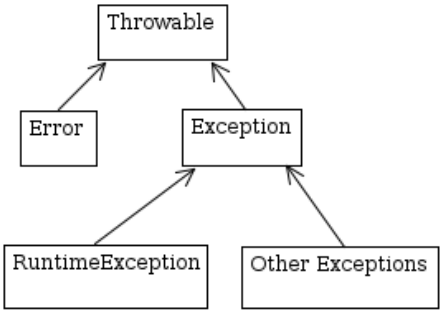
```
Employee[] ea=new Employee[100];
//ea is populated from a database with a mixture of subclass objects
for(int i=0;i<ea.length;i++)
{
  ea[i].applyRaise(.03f);
}
```

Polymorphism will take care of making sure that the correct method is called for the different subclass objects. This is a way of programming **generically**.

The power of this is that I can add new subclass objects later and I don't need to modify the above code. After compiling polymorphism will take care of everything.

Unit 3.6 – Exception Handling

The superclass for all exception classes is **Throwable**. Under that are two subclasses **Error** and **Exception**. Under Exception is **RuntimeException** and other exception classes.



Normally we don't bother with the Error classes and subclasses. They handle things like divide by zero, etc. BTW, you should always check your variables so that you don't divide by zero, but you should never need to use the Error classes in your code.

Checked and Unchecked Exceptions

Exception classes are divided into two categories **checked** and **unchecked** exceptions. The "checking" refers to the compiler. RuntimeException and its subclasses are unchecked exceptions. The compiler doesn't care if you handle them or not. Checked exceptions must be handled or your code won't compile.

An example of an unchecked exception is NumberFormatException (NFE). Suppose you called the parseInt method of the Integer class and passed the string "abc". The JVM will throw an NFE. This might cause problems with your program, so you should handle it anyway, but you are not required to, and it will compile without handling.

An example of a checked exception is FileNotFoundException. Suppose you tried to read a file that didn't exist. That would be a problem. In this case your program wouldn't compile without exception handling.

If you create new exception classes, you can make them unchecked by extending RuntimeException. If you extend Exception then your new classes will be checked exceptions. We will look at examples of this later.

Using Try-Catch Blocks

The most common way of handling exceptions is with a try-catch. The syntax looks like this:

```
try
{
  ...  //code that throws WhateverException
}
catch(WhateverException we)
{
  ...
}
```

A few notes on this. It is common to name your variable using lower-case letters for the capitalized letters of the exception class. In this case the made-up name WhateverException would have a variable "we". IOException would be "ioe". This is just a convention and you don't need to follow it.

<rant>One of my pet peeves is that if none of the code in the try block throws the exception in the catch the compiler will barf up an error message. I use comments a lot when I'm debugging. I'll often comment out a whole block of code when things aren't working. If I comment out the call that throws the exception then the compiler will yell at me with a message like "exception Whatever Exception is never thrown in the body of the corresponding try statement". So then I have to comment out the try-

catch as well, which is extra, unnecessary work. If it doesn't throw an exception then don't catch it!**</rant>**

Another issue is what to do in the catch block. At minimum you should print the exception to the console. Otherwise you might not be aware that the exception occurred. Simply do this:

```
System.out.println(we);
```

You may want to do other things as well. For example, when you get a NumberFormatException you might want to set your variables to some default value so that you recover gracefully from the exception.

Multiple Catch Blocks

A try-catch can have more than one catch block. For example, you may have code that does a parseInt and a file read or write in the same block. Clearly you want to handle those differently.

There is one thing to be aware of though. The first catch that matches will be processed. Consider the following code:

```
try
{
    ...
}
catch(IOException ioe)
{
    ...
}
catch(FileNotFoundException fnfe) //don't do this!
{
    ...
}
```

Since FileNotFoundException is a subclass of IOException, any FNFE is also an IOE, so the second catch is useless. It will never be executed. Always put the subclasses first (go from specific to general).

Throwing Exceptions with the "throws" Keyword

As an alternative to the try-catch, you can also mark your methods to throw the exception. Then the compiler will not issue an error message if you don't handle the exception locally. However, when that method is called it will check to see if it is handled or thrown (if it is a checked exception).

Consider the following example:

```
public void methodA() throws WhateverException
{
  ...  //code that throws WhateverException
}
```

Note the absence of a try-catch. This basically moves the responsibility of handling the exception from methodA to the code that calls methodA.

According to Prof. Barker you can use the throws keyword to throw the exception all the way back to the JVM.

Some context for this: when writing data structures (CM307) is is very common to throw exceptions. When writing applications (CM335) you typically handle the exceptions locally as they occur. Think of it as "the buck stops here".

Throwing Exceptions with the "throw" Keyword

This applies more to data structures than applications, but for the sake of completeness I will mention it here. If you want to throw an exception you do it like this:

```
if(something bad happens) throw new WhateverException();
```

Note that the keyword "**throw**" is different from the keyword "**throws**" and they are used very differently.

Creating new Exception Classes

Exception classes are actually very simple. They are a wrapper for a message (a String object). To make a new unchecked exception simple extend RuntimeException like this:

```
public class MyException extends RuntimeException
{
  public MyException()
  {
    super("default message");
  }

  public MyException(String s)
  {
    super(s);
  }
}
```

If you want a checked exception then extend Exception instead of RuntimeException.

Note that in this case there is a default message (String) or the programmer can pass a specific message when they create the exception object.

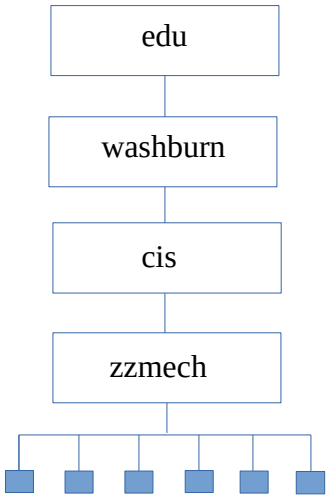
I always thought it would be fun to create an exception class IncompetentUserException and throw it when a user puts letters into a box that requires numbers.

Unit 3.7 – Packages

Packages are a way of organizing Java classes. They are commonly used for third-party code written by vendors to support their products. They are usually delivered through **jar files**. Jar = Java archive.

<**aside**>The idea of the jar file comes from the UNIX **tar file** (tar = tape archive). In the old days UNIX tape backups stored tar files that contained whole directory trees that were serialized into a single file that could be stored on a tape backup unit.</**aside**>

Packages have a hierarchy that must match a directory hierarchy. It's typical to use the organization name and department in the hierarchy. Consider the following:



class files

The package name for this directory structure would be edu.washburn.cis.zzmech. The source code for the class files would all start with:

```
package edu.washburn.cis.zzmech;
```

When an application programmer uses this package they would include the following in their source code:

```
import edu.washburn.cis.zzmech.*;
```

There's a little more to it of course. When importing packages that aren't part of the Java library you

must also use a **classpath** environment variable or add the directory tree or jar file to your IDE. I always compile and run on the command line (and I'll ask you to do the same for Project 1), so I'll talk about that here.

If you use a package often it is probably worthwhile to set your system classpath environment variable so that it is always there. (I won't talk about how to do that here, you can find many Google links to tutorials on that topic.) If it's something you only do once in a while, you can set the classpath on the command line. Unfortunately the syntax depends on your operating system. I'll show you how to set the classpath on the command line in Linux/UNIX/Mac and Windows.

Let's say my application program name is Driver1.java and it needed to use one or more class files in the package edu.washburn.cis.zzmech. I would compile and run like this in Linux/UNIX/Mac:

```
javac -cp .:/home/zzmech/cm203/project1 Driver1.java
java -cp .:/home/zzmech/cm203/project1 Driver1
```

and like this in Windows:

```
javac -cp .;C:\Users\zzmech\cm203\project1 Driver1.java
java -cp .;C:\Users\zzmech\cm203\project1 Driver1
```

The path after the colon or semicolon is the path to the edu directory, the top of the package directory tree. The "." before the colon or semicolon is included so that the JVM looks in the current directory for any class files that it needs. For some reason if you don't set the classpath it will always look in the current directory. But if you specify the classpath it won't, unless you explicitly include it.

You'll do all of this in Project 1.

Packages in Jar Files

In order to create a jar file on the command line, go to the directory that contains the "edu" directory and type this command:

```
jar -cvf jarfilename.jar edu
```

The "c" means create, "v" means verbose, "f" means the filename follows. You can use "-xvf" to unjar the file and "-tvf" to list the contents of the jar file. BTW you can use the jar command to create zip files, just use the extension .zip instead of .jar.

When compiling and running on the command line using jar files, you need to include the jar file name in the classpath:

```
javac -cp .:/home/zzmech/cm203/project1/jarfilename.jar Driver1.java
java -cp .:/home/zzmech/cm203/project1/jarfilename.jar Driver1
```

and

```
javac -cp .;C:\Users\zzmech\cm203\project1\jarfilename.jar Driver1.java
java -cp .;C:\Users\zzmech\cm203\project1\jarfilename.jar Driver1
```