Abstract Data Types

Good programming practice to wrap up a type with methods to manipulate that type.

Facilitates *software reuse.*

Modern languages have *libraries* to further facilitate software reuse.

New classes can be added easily.

Problem (seen in Practical 2)

```
LinkedList a = new LinkedList();
a.add("Bill");
a.add("Ben");
String s = (String) a.get(1);
   // Note use of cast

a.add(2);
Integer i = (Integer) a.get(2);
   // another cast
```

What about

```
String s = (String) a.get(2);
```

Generics or Parameterised Types

Solves the problem of casting, and run-time type errors.

Instead of:

```
LinkedList a = new LinkedList();
```

we write:

```
LinkedList<String> a = new
  LinkedList<String>();
```

(String is just an example here, it could be any other type)

If we try to add other kinds of object to the list, we get a compile time error; hence, we get more language support for reliable programming.

We will no longer need to use a cast when using `get`.

`LinkedList<E>` and `ArrayList<E>`

Two classes providing similar methods. E.g.

```
boolean isEmpty()
int size()

E get(int index)
E set(int index, E elem)

boolean add(E elem)
E remove(int index)
```

See the Java API to discover more about these classes and methods.

Benefits of ArrayList and LinkedList

Notice that all the ways of accessing elements and adding/removing them remain the same for both classes; therefore it is easy to switch between them – we only have to change the line declaring the object.

This is a good example of information hiding: we can use lists, without knowing implementation details (is it an array, or a linked list).

`LinkedList` and `ArrayList`

Which to choose?

There may be efficiency differences depending on the application. Items to consider include:

- adding or removing an item from the middle (and consider different ways to do this)
- adding a new element to the end
- what if the capacity of the list is reached?

See the Java API for more detail on algorithmic efficiency.

`List` interface

Remember that an **`interface`** is like a very abstract class; all its methods are abstract.

We ***extend*** a class, but ***implement*** an interface.

The Java library has a **`List`** interface.

Both **`ArrayList`** and **`LinkedList`** implement the **`List`** interface.

Example of use

We can define either:

```
List<String> a = new ArrayList<String>();
```

or:

```
List<String> a = new LinkedList<String>();
```

This gives us even more support for re-usable code.

We only commit to an implementation when a new object is created.

Another problem

Example:

```
public void printList(List<String> a) {
   for (int i = 0; i < a.size(); i++)
      System.out.print(a.get(i) + " ");
} //printList
```

What does this code do?

Do you think it will be faster with **`ArrayList`**, or **`LinkedList`**? Or the same with both?

Iterator

We can forget all about the underlying implementation:

An **`Iterator`** is an object which enables us to visit all the elements in a collection of objects.

With lists, the order is obvious and we can use a **`ListIterator`**.

What if there is no order? If there's no order, we don't care, as long as every element is visited.

ListIterator

Example

```
public void printList(List<String> a) {
    ListIterator<String> iter = a.listIterator();
    while (iter.hasNext())
          System.out.print(iter.next() + " ");
} //printList
```

ListIterator

Another example:

```
public void toUpper(List<String> a) {
    ListIterator<String> iter =
   a.listIterator();
    while (iter.hasNext()) {
        String s = iter.next();
        s = s.toUpperCase();
        iter.set(s);
    }
} //toUpper
```

Patterns

Use of iterators conforms to a ***standard pattern***:

```
public void anOperation(List<String> a) {
    Iterator<String> iter = a.iterator();
    while (iter.hasNext()) {
      String temp = iter.next();
      // perform operations on current
  String
    }
} //anOperation
```

Collections

Further up the hierarchy: a more general interface called **`Collection`**.

The **`List interface`** extends **`Collection`**.

The **`Set interface`** also extends the **`Collection`** interface.

The **`Set`** interface is implemented by **`HashSet`** and **`TreeSet`** (for example).

An iterator is essential if we are to visit all the elements in a set because there is no ordering of set elements.

Linking `Collections` to `Arrays`

Among the methods in class **`Arrays`** are:

```
static int binarySearch(int[] a, int key)

static void sort(int[] a)
```

There are overloaded versions of these methods for, among others, **`double`** and **`Object`**.

There is also the method:

```
static List asList(Object[] a)
```

This allows us to ***view*** an array of objects as a fixed size **`List`** and to perform **`List`** methods.

We must not, however, use the methods **`add`** or **`remove`**.

Examples

Example 1

```
int vals[] = {7, 18, 9, 23, 56, 12};
Arrays.sort(vals);
int position = Arrays.binarySearch(vals,
  23);
```

Example 2

```
Integer ivals[] = new Integer[6];
for (int i = 0; i<6; i++)
   ivals[i] = new Integer(vals[i]);
List<Integer> vList;
vList = Arrays.asList(ivals);
vList.set(2, new Integer(19));
```

`Collections`

This class is similar to **`Arrays`** except that it contains static methods to deal with classes that implement the **`Collection`** interface.

Example:

```
Collections.sort(vList);
```

Generics Revisited

We can define our own ***parameterised type***.

```
public class Stack <Kind> {
  private List<Kind> val;

  public Stack() {
    val = new ArrayList<Kind>();
  } // constructor

  public void push(Kind x) { ... }
  public Kind pop() { ... }
  public boolean isEmpty() { ... }
} // Stack
```

Generics

<Kind> is like a formal parameter. As long as we are consistent, we can use any name.

We can now declare a stack of **`String`** objects as:

```
Stack<String> aStack = new Stack<String>();
```

We only have to define (and test!) one **`Stack`** class: generics ensure that it will work for any kind of object.

We have full type checking at compile time.

Kind can be any type we want, including ones we've defined ourselves.

Inheritance

We start with a ***superclass*** from which a ***subclass*** can be derived.

New attributes and methods can be defined in a subclass and existing methods re-defined.

How do we know which version to use?

We shall look at inheritance and *dynamic binding* in C++ and Java.

In C++, a superclass is known as a ***base class*** and a subclass is known as a ***derived class***.

Java class Person

```
public class Person {
  protected String theName;

  public Person(String  name) {
    theName = name;
  } // constructor

  public String getInfo() {
    return theName;
  } // getInfo

  public String detailInfo() {
    return "Details are: ” + this.getInfo();
  } // detailInfo
} // Person
```

Calls

Java:

```
Person woman = new Person("Sue");
String st = woman.getInfo();
```

C++:

```
Person man("Jim");
Person* woman = new Person("Sue");
printf(woman->getInfo());
```

Java class `Student`

```
public class Student extends Person {
  private int theRegNum;

  public Student(String name, int reg) {
    super(name);
    theRegNum = reg;
  } // constructor

  public int getRegNum() {
    return theRegNum;
  } // getRegNum

  public String getInfo() {
    return theName + "," + theRegNum;
  } // getInfo
} // Student
```

Inheritance

Advantages:

- software re-use
- faster software development

Inheritance should only be used when there is a ***logical relationship*** between a superclass and a subclass.

This is the ***is_a*** relationship.

We have: Student ***is_a*** Person

Inheritance: type checking

An object of a subclass can always be used when an object of its superclass is expected.

Consider the following Java declarations:

```
Person p1 = new Person("Fred");
Person p2 = new Student("George",974503);
```

The ***type*** of both `p1` and `p2` is *reference to* **`Person`**.

Inheritance

What happens in the call:

```
String st = p2.getInfo();
```

Which definition of **`getInfo`** is called?

As **`p2`** is currently pointing at a **`Student`** object, it is the definition of **`getInfo`** defined in class **`Student`** that is called.

So we will produce name and reg number.

Dynamic binding

Consider now the call

```
p2.detailInfo();
```

The only definition of **`detailInfo`** is in class **`Person`** and so that is the method that is called.

During its execution we have the call

```
this.getInfo();
```

What is the result?

As **`p2`** is currently pointing at a **`Student`** it is the version of **`getInfo`** defined in **`Student`** which is executed.

Type of `p2`

Consider now the call:

```
p2.getRegNum();
```

Method **`getRegNum`** is defined in class **`Student`** and **`p2`** is currently pointing at a **`Student`** object.

What result does this have?

The ***type*** of **`p2`** is *reference to* **`Person`** and so this call is not legal and would give a ***syntax error***.

Overloading v Redefinition

In Java, when a method is redefined in a subclass with exactly the same parameter profile as the original method in the superclass then we have ***redefinition*** and the binding of method calls occurs at run time, i.e. it is ***dynamic***.

If the new method has the same name, but a different parameter profile, then we have overloading and the binding occurs at compile time, i.e. it is ***static***.

Overloading

Two procedures, functions or methods in C++, Java or Ada may have the same name and different parameter profiles:

```
void swap(int& a, int& b) { ... }

void swap(double& a, double& b) { ... }
```

Separate code is generated for each of the overloaded routines, and the compiler decides ***at compile time*** which one to use.

We therefore have ***static binding***.

Overloading

In Java, we often have several constructors:

```
class BankAcc {
   …
   public BankAccount(String id) { …}
   public BankAccount(String id, int amount) {…}
   …
}
```

This is an example of overloading.

Redefinition in C++

C++ supports both static and dynamic binding of methods.

When a method is defined in the superclass to be **`virtual`** as in:

```
virtual String getInfo() { ... }
```

then it may be redefined in a subclass giving us dynamic binding.

If the reserved word **`virtual`** is omitted then the binding is ***always*** static. This is more efficient.