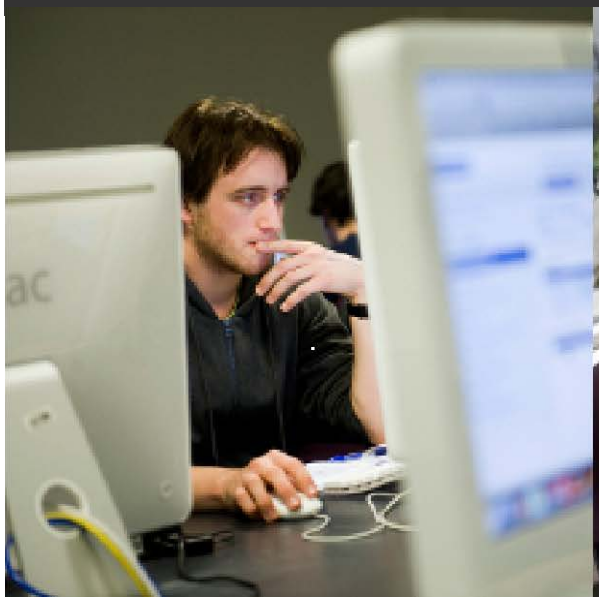
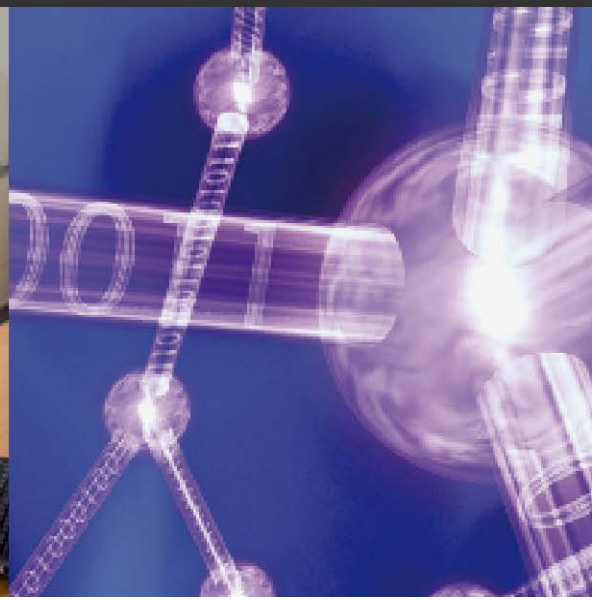
MONASH University

Information Technology

Module 6:
Arrays, ArrayLists and Aggregation

FIT2034 Computer Programming 2
Faculty of Information Technology





Part 1:
Arrays

Grouping Data

- We often want to deal with many related variables all of the same data type
  - e.g. daily sales figures for a year
    - The data type here is double (i.e. a primitive type)
  - e.g. all bank accounts objects in a banking application
    - The data type here is a reference type
- Possible solution

```
double day1;
double day2;
:
:
double day366; //allow for leap year
```

366 variables declared

```
total = day1 + day2 + … + day366;

day1 = day1 - (0.1 * day1);
day2 = day2 - (0.1 * day2);
:
day366 = day366 - (0.1 * day366);
```

  - What if we need to add these daily sales figures to get a yearly total?

366 terms

  - What if we need to subtract tax from each daily sales figure?

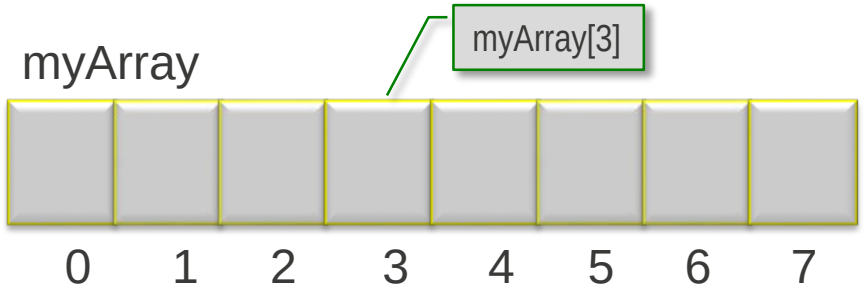
366 statements

Grouping Data - What is Required

- Rather than creating a variable for each value separately we require a “data structure” that:
  - Can store any number of variables of the same type
  - Can be referenced as a whole by a single name
  - Allows access to the individual variables (called elements) in the structure
    - Using the structure’s name and an index
    - So all variables have the same name but differ by an index
    - This index could then, for instance, be incremented in a repetition control structure to process all elements of the data structure

Java Arrays

- Java has such a data structure that meets all these requirements
  - It's called an **array**
- Notes
  - The array has a name
    - It's the name of its reference variable (yes it's an object)
  - It has component elements (or members) each uniquely identified by an index/subscript number (which start at 0)
    - These are stored contiguously (next to each other in (heap) memory)
    - Each component is a variable
  - An array's size (number of elements) is specified when it is instantiated - It cannot be subsequently resized



Declaring Arrays

- Arrays are declared by placing open and close square brackets after either the data type, or the name
- e.g.
  int [ ] marks;
  - Declares ‘marks’ as a reference variable that can reference (point at) any array of integers no matter what size (number of elements)
  - Initially it points at no array object
    - No array has been instantiated or assigned to it yet

Instantiating Arrays

- We must specify how many elements (places) there are going to be, using a 'new' statement
- Example:

```
marks = new int[10];
```

  - An array of integer with 10 elements (index 0 → 9) is instantiated ('new' finds the necessary heap memory)
  - 'new' returns the array's memory address which is then assigned to the reference variable marks
    - marks can still reference any array of integers no matter what size – not only those of size 10 (i.e. those with 10 elements)
- Both the declaration and instantiation can be done in one statement:

```
int[] marks = new int[10];
```

Array Initialiser Lists

- Normally, all elements will be given a default value
  - 0 for numeric types
- Java has syntax to allow us to initialise the values of an array's elements as the array is declared
  - It's only really practical for arrays with a small number of elements
  - It's called an initialiser list
  - e.g.

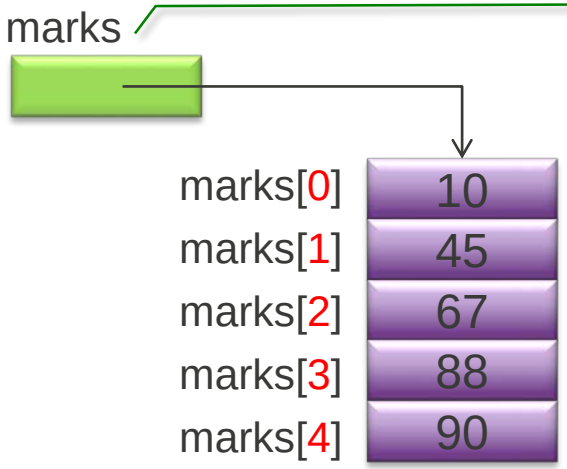
```
int[] marks = {10, 45, 67, 88, 90};
```

Curly brackets indicate start and end of a comma-separated list of values

    - Notes
      - There is no instantiation operator but an array object is instantiated
      - The array's size (number of elements) is NOT specified as the compiler can determine this from the initialiser list

Accessing Array Elements

- e.g. `int[] marks = {10, 45, 67, 88, 90};`



The reference variable marks references an array of int with 5 elements.
Note: it could be reassigned to point at any array of int regardless of number of elements of int in the new target array.
Note: it actually points at the first element of the array since the run-time system can quickly access all other elements since their memory locations are easily calculated as offsets from this first element.

- To access an array element
  - `arrayName[elementIndex]`
    - Index is in the range 0 → array length – 1
      - Where array length is the number of elements in the array
    - The maximum index of any array is always its length – 1 (a consequence of zero based indexes)

Array Indexes

- Array indexes (inside the square brackets):
  - Can use any expression that evaluates to an integer value
  - Are also known as subscripts
- All of the following compiles without errors:

```
int subscript = 0;
Scanner console = new Scanner(System.in);
int myMark;

int[] marks = {10, 45, 67, 88, 90};

myMark = marks[3];
myMark = marks[subscript + 2];
myMark = marks[console.nextInt()];
```

- ArrayIndexOutOfBoundException
  - If during execution an attempt is made to access an array at an index outside of the range 0 → array length -1
    - an ArrayIndexOutOfBoundException will occur and execution will stop

Set/Get Array Element Values

- Set `arrayName[elementIndex] = expressionOfTheCorrectType`
  - e.g.

```
marks[3] = 100;
```

- Get `variableOfTheCorrectType = arrayName[elementIndex]`
  - e.g.

```
myMark = marks[0];
```

- An array element can appear syntactically wherever a variable of the same type can appear
  - e.g.

```
System.out.println((double)marks[2] / 5);

total = total + marks[i];
```

Arrays in Use

- The following code uses an array to translate an integer input by a user to a day of the week
  - It does this by using the integer as a “look-up” index into the array

```
String[] dayName = {"Sunday", "Monday",
                    "Tuesday", "Wednesday",
                    "Thursday", "Friday",
                    "Saturday"};

Scanner console = new Scanner(System.in);
int dayOfWeek = console.nextInt();

if ((dayOfWeek < 1) || (dayOfWeek > 7))
   System.out.println("Invalid day: Enter a value from 1 to 7.");
else
   System.out.println("Today is " + dayName[dayOfWeek - 1]);
```

Note the “off-by-one” adjustment:
From a 1 based, human-friendly index (1 → 7) to a zero based, Java-friendly index (0 → 6).
An alternative is to use an array with 8 elements and never use the 0^{th} index element

Arrays and while … Loops

```
int[] marks = {10, 45, 67, 88, 90};
int position = 0;

while (position < marks.length){
    System.out.println(marks[position]);
    position++;
}
```

```
10
45
67
88
90
```

- Notes
  - marks.length
    - Every array has a public constant called length that is automatically set to contain the number of elements in the array
    - The highest index of an array is its length – 1
    - Compare an array's length constant to a String's length() method (methods always have trailing brackets even if they are empty of parameters)
  - Alternative loop condition to the example's condition:
    - Some programmers prefer

```
while (position <= marks.length - 1){…
```

  - The code would be the same no matter how big the array (10,000 elements! More!)
    - Arrays + repetition control structures = power

Arrays and for … loops

- for … Loops and arrays are made for each other
  - For loops are the preferred repetition structure when the number of repetitions is known
  - When processing an array the number of repetitions is known (0 → length – 1)
- e.g. Solving our daily sales problems discussed earlier:

```
double[] dailySales = new double[366];
double total = 0;

//assume all elements of daily sales are set somehow

for(int i = 0; i < dailySales.length; i++)
    total = total + dailySales[i];

for(int i = 0; i < dailySales.length; i++)
    dailySales[i] = dailySales[i] - (0.1 * dailySales[i]);
```

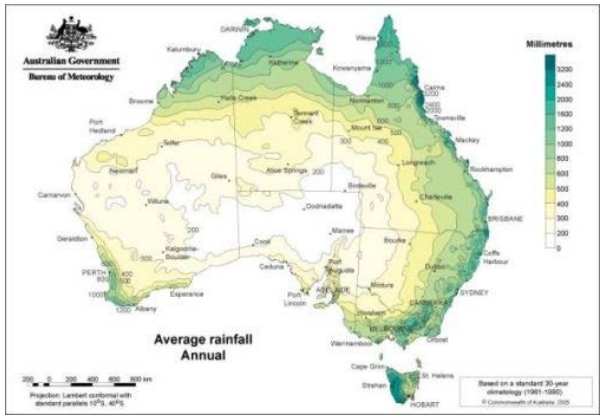
alternative condition would be: i <= dailySales.length - 1

sums all elements of dailySales

removes tax of 10% from all elements of dailySales

Example: Monthly Rainfall

- Problem
  - Using this rainfall table write Java code to calculate the mean monthly rainfall for the year



Month	Rainfall (mm)
January	30
February	40
March	45
April	95
May	130
June	220
July	210
August	185
September	135
October	80
November	40
December	45

Example: Monthly Rainfall

Average monthly rainfall:104.58333333333333

```
public class MonthlyRainfall{
   public static void main(String[] args){
      double totalRainfall = 0, meanRainfall = 0;

      // Declare a monthly rainfall array (12 elements: indexes 0 .. 11)
      int[] rainfall = {30, 40, 45, 95, 130, 220, 210, 185, 135, 80, 40, 45};

      for (int m = 0; m < rainfall.length; m++)// Iterate through months
         totalRainfall += rainfall[m];        // Calculate total rainfall

      // Calculate the average monthly rainfall
      meanRainfall = totalRainfall / rainfall.length;

      // Display the average monthly rainfall
      System.out.print("Average monthly rainfall:" + meanRainfall);
   }
}
```

No need to protect against division by zero in average calculation as denominator is rainfall.length > 0

Testing Bounds

- To be on the safe side, always test that array indexes are within bounds, especially if they come from user input.

```
import java.util.*;

public class RainfallBounds {
   public static void main(String[] args){
      // Declare a monthly rainfall array (12 elements: 0 .. 11)
      int[] rainfall = {30, 40, 45, 95, 130, 220, 210, 185, 135, 80, 40, 45};

      int month=0;
      Scanner console = new Scanner(System.in);

      System.out.print("Enter month [1-12] for rainfall output: ");
      month = console.nextInt();
      while (month < 1 || month > 12){
         System.out.println("Incorrect Input. Try again.");

         System.out.print("Enter month [1-12] for rainfall output:");
         month = console.nextInt();
      }
      System.out.print("Rainfall in month #" + month + " is " +
                       rainfall[month - 1]);
   }
}
```

```
Enter month [1-12] for rainfall output: 22
Incorrect Input. Try again.
Enter month [1-12] for rainfall output: 5
Rainfall in month #5 is 130
```

while loop can only be exited if valid user input is provided

must be a valid value since loop has exited

off-by-one adjustment

Equality of Arrays

- Arrays are objects
  - Therefore testing for equality (==) between two array object references tests equality of addresses
    - i.e. it tests if the references are aliases
- To test if the contents of two independent arrays are equal we must test element-by-element using a loop

```
int[] rainfall2007 = new int[12];
int[] rainfall2008 = new int[12];

// Suppose some code here sets all the values in arrays for 2007
and 2008

boolean contentsEquivalent = true;
for (int i = 0; i < 12; i++)
   if (rainfall2007[i]!= rainfall2008[i])
      contentsEquivalent = false;

if (contentsEquivalent)
   System.out.println("It rained the same amount every month!");
```

Assume, initially, they are equal

If all the corresponding elements (i.e. those with the same index) have the same rainfall then this if condition will never be true and contentsEquivalent will remain untouched i.e. keep the value true.
If one or more corresponding elements have different rainfalls contentsEquivalent will be set to false. Q: How efficient is this if it changes early?

Array Processing

- The most fundamental array processing task is to process each element in the array
  - We have already seen how to perform such a process
    - Using a while … loop and a for … loop and
    - Incrementing an array's index progressively from 0 up to the array's length – 1 using an integer variable
- This element-by-element processing can be used to perform many array processing tasks including:
  - Changing each element's value in the array
  - Linear (Sequential) Search tasks
    - i.e. searching for a particular value or values that satisfy some criteria or a maximum or minimum value and reporting these values and possibly the index(es) they were found at
    - All of these involve processing an array from its first element until its last (in order) or until a required value is found

Array Processing - Examples

- Reconsider the monthly rainfall table
  - Implemented as an array
- The following can all be classified as array search problems:
  - Determine when (or if) the rainfall was exactly 95 mm
  - Determine how many times it was over 100 mm
  - Determine the highest/lowest monthly rainfall
  - Determine when the highest/lowest rainfall occurred

Month	Rainfall (mm)
January	30
February	40
March	45
April	95
May	130
June	220
July	210
August	185
September	135
October	80
November	40
December	45

Example: Linear Search

- Problem
  - In which month was the rainfall 95 mm?
- Algorithm
  - Process each element from the first (index = 0) to the last (index = length – 1) until 95 mm is found or the last element is processed (95 mm not found)
  - i.e. a linear search

This is just narrative English. Pseudocode and code follow on the next slides.

Month	Rainfall (mm)
January	30 ✗
February	40 ✗
March	45 ✗
April	95 ✓
May	130
June	220
July	210
August	185
September	135
October	80
November	40
December	45

Example: Linear Search - Pseudocode

```
set valueToFind, found = false, index = 0

while (not found AND elements left to process)
      if (thisElement's value = valueToFind)
            found = true
            indexToFind = index

      increment index

if  (found)
      display found message including indexToFind

else
      display not found message
```

Note: this algorithm only finds the first instance of 95. The one with the lowest index. Other subsequent occurrences of 95 are never even processed.

This is pseudocode (not Java code) Remember this is structured English with a relaxed syntax designed to help us focus on logic not Java language features and syntax.

Example: Linear Search Code

```
int[] monthyRainfall = {30, 40, 45, 95, 130, 220, 210, 185, 135, 80, 40, 45};
int valueToFind = 95;
int indexOfValueToFind = -1;

boolean found = false;
int i = 0;

while (!found && (i < monthyRainfall.length)){
    if(monthyRainfall[i] == valueToFind){
        found = true;
        indexOfValueToFind = i;
    }
    i++;
}

if(found)
    System.out.println( "first instance of " + valueToFind + " is at index " +
                        indexOfValueToFind);
else
    System.out.println("no instance of " + valueToFind + " found");
```

We could get the value to search for from user input for a more interesting program

?

Either we find it and we exit the loop, OR
We get to the last element without finding it and we exit the loop

We should save the current value of the index now before its incremented again so we have the correct index after the loop exits

After the loop has exited we use the value of found to determine why the loop exited

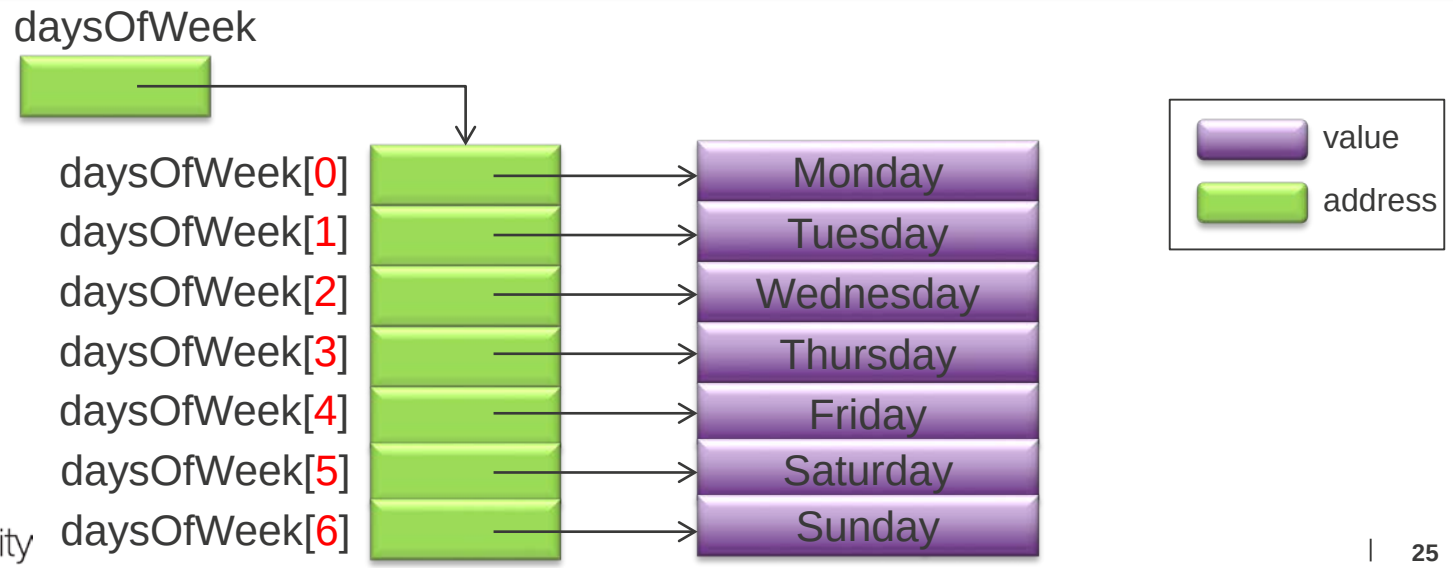
first instance of 95 is at index 3

Arrays of Object References

An Array of String Object References

- It's common to hear programmers talk of an array of Strings or an array of some other reference type
  - This is a convenient shorthand but what they really mean is an array of object references (not the objects themselves)
- e.g.

```
String[] daysOfWeek = { "Monday", "Tuesday", "Wednesday",
                        "Thursday", "Friday", "Saturday",
                        "Sunday"};
```



Arrays of Object References …

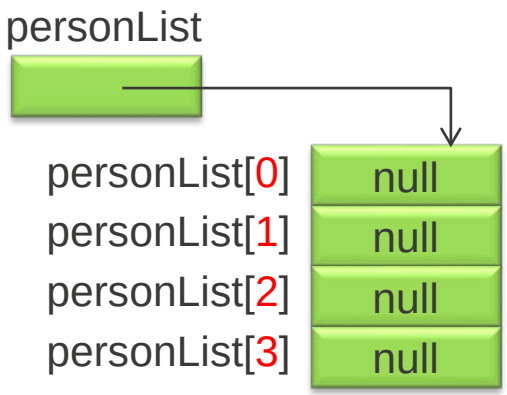
- Definition of the Person class
  - Note the use of the mutator in the class's own code to centralise validation
  - This class is used in next few slides

```
public class Person {
 private int age;
 private String name;

 public Person(int initAge, String initName){
   setAge(initAge);

   name = initName;
 }

 public Person(int initAge){
   setAge(initAge);
   name = "nobody";
 }

 public void setAge(int newAge){
   if (newAge > 0)
     age = newAge;
   else
     age = 0;
 }

 public int getAge(){
   return age;
 }

 public void setName(String newName){
   name = newName;
 }

 public String getName(){
   return name;
 }
}
```

Arrays of Object References

- Instantiating an Array of Object References does *not* instantiate any objects
  - Recall: uninitialised object references:
    - Are set to null if the object reference is an instance variable
    - Are left uninitialised if the object reference is a local variable
  - So: Array elements that are object references are automatically initialised to null
    - Whether instance or local variables
- e.g.

```
Person[] personList = new Person[4];
```

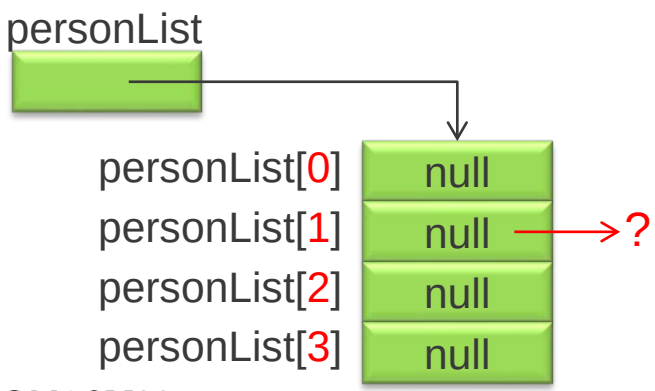


  - 4 reference variables capable of referencing Person objects have been declared as elements of the personList array
  - NOTE: No Person objects have been instantiated

Arrays of Object References

- The following code will stop execution and a NullPointerException occurs
  - Because instantiating an array of object references does not instantiate the objects
  - Basically the exception means there is no reference to a Person object to get the age from

```
Person[] personList = new Person[4];

System.out.println(personList[1].getAge());
```



personList[1] is just a reference variable of type Person so its syntactically valid to invoke the getAge accessor on it using dot after the close square bracket.

Remember: An array element can appear syntactically wherever a variable of the same type can appear.

The problem here is not syntax, but rather that currently personList[1] = null.

Arrays of Object References

- Before we can use the object references in an newly instantiated array of object references:
  - We need to instantiate objects of the appropriate type (or use existing ones) and then reference them with the array element object references

```
Person[] personList = new Person[4];

personList[0] = new Person(20, "John");
personList[1] = new Person(22, "Helen");
personList[2] = new Person(24, "Cynthia");
personList[3] = new Person(18, "Holly");

System.out.println(  personList[1].getAge() + " " +
                     personList[1].getName());

for (int i = 0; i < personList.length; i++)
    personList[i] = new Person(0);

System.out.println(  personList[1].getAge() + " " +
                     personList[1].getName());
```

Here the Person class's 2 parameter constructor is being used to instantiate Person objects and initialise each object's state with CUSTOM data.

personList[i] references a Person object and can therefore have Person methods invoked on it without error.

Here a for … loop and the Person class's 1 parameter constructor is being used to instantiate Person objects and initialise each object's state with the SAME data. This is often all we want to do initially (to ensure non-null elements).

Part 2:
The ArrayList class

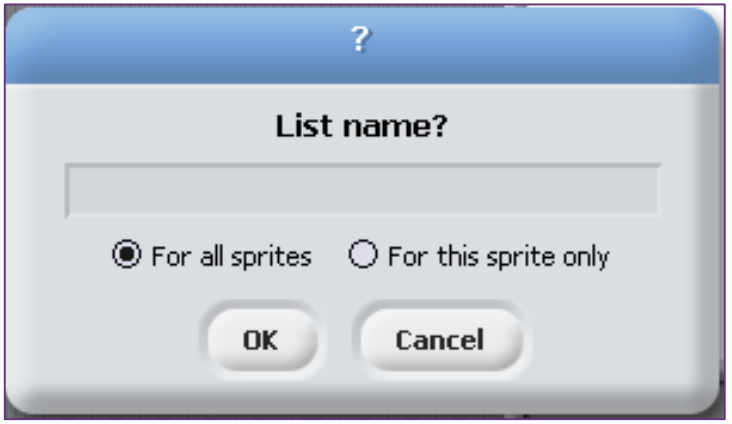
Objectives – Part 2

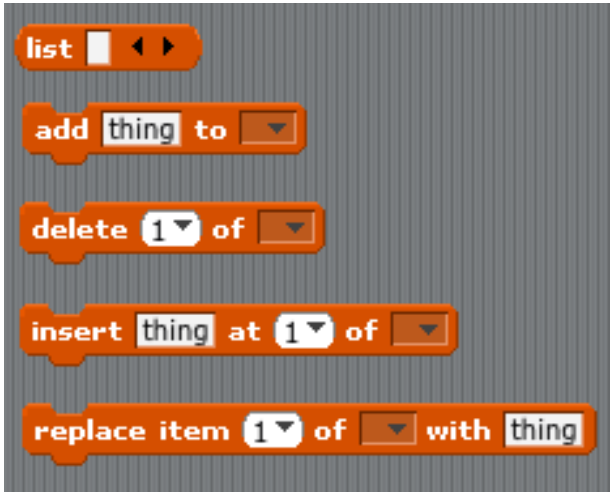
- Declare, instantiate and use ArrayList objects to manage a collection of data items;
- Make appropriate use of wrapper classes; and
- Write code to implement aggregation relationships using arrays or ArrayLists;
- Describe the role of and be able to implement a data management class;

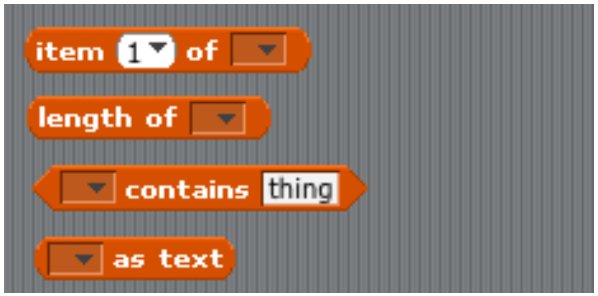
The ArrayList Class

- The ‘ArrayList’ class is a class from which you can make instances (objects).
- The instances are responsible for managing a set of objects of a common type, and therefore are like an array,
  However…
- You can dynamically add more objects into and remove objects from the ArrayList
  - So they are like Scribble’s Lists
  - You can also remove objects from the ArrayList.
- Objects are identified by their index/position
  - Indexes begin at 0 in Java

List commands and operators from *Scribble*







List Example from Scribble

- Later slide shows Java equivalent code
  - Add places at end of list
  - Insert puts before existing element at stated index
    - Scribble index begins with 1
    - Pear goes before apple
    - Apple is first said, then deleted from list

Using ArrayList

- **Declaration** of an ArrayList object requires you to specify the types of objects to go in the list:

```
ArrayList<Dog> myPets;
```

The array list is only able to contain references to Dog objects.

- **Instantiation** of the object also requires the special syntax:

```
myPets = new ArrayList<Dog>();
```

Don't forget the round brackets

You must include the type of objects again when using 'new'

Don't forget the round brackets always required when instantiating references

Using ArrayList

- Initially, an ArrayList object contains no elements – it is empty.
- **Adding** elements – supply the object you wish to add, by calling the 'add' method.

  myPets.add(fido);
  - adds 'fido' to the end of this array

  myPets.add(4, woofer)
  - inserts at 5th position. Moves old 5th to 6th position, old 6th to 7th.
- **Retrieving** elements (leaving them in the list)

  myPets.get(3);

  - retrieve Dog which is in 4th position.
- **Removing** elements – use index of one to remove

  myPets.remove(1);
  - remove Dog in position 2, shifts position 3 to now be position 2

Using ArrayList

- To find out how many items currently are in the ArrayList's collection of objects, you can use the size() method

  System.out.println(myPets.size());

  - Reports the number of objects currently in the ArrayList

- The index of a particular element may change if you insert or remove from the middle of the list.
  - All items to the "right" will move one spot so that all indexes remain relative with no gaps
  - The size() will be reduced

The Example converted to use Java ArrayLists

```
ArrayList<String> words;
words = new ArrayList<String>();

words.add("apple");
words.add("banana");
words.add(0,"pear");       // Indexes begin at 0 in Java
System.out.println(words.get(1));
words.add("watermelon");
words.remove(1);

if (words.contains("apple"))
    System.out.println("I will eat an apple");
else {
    System.out.println("Instead I will have to eat a " +
        words.get(0)
    );
}
```

Wrapper Classes

Wrapper Classes

- Sometimes a primitive type variable needs to behave as if it is an object.
- This may be because
  - We need the variable to exhibit some smart behaviour, such as converting itself to a different format, or
  - passing it as a parameter to a method which expects objects as parameters.
- For each primitive data type, there is a corresponding wrapper class:
  - **Integer**, **Short**, **Long**, **Float**, **Double**, **Character**, **Byte** and **Boolean**
- All are **`public final`**, meaning they cannot be extended (explained next week)
- they are used to create objects from their corresponding primitive types

Wrapper Classes

- Each primitive type has an associated class that can be used when real objects are needed.
- For example (incomplete):

primitive type	corresponding class type
`int`	`Integer`
`char`	`Character`
`boolean`	`Boolean`
`double`	`Double`
`short`	`Short`

Note the use of Capital letters for the class types, but lowercase for the primitive types

Wrapper Classes

Every wrapper class has 2 forms of constructors :

- one takes the primitive type and returns an object of the wrapper type

  Integer(5) returns an **`Integer`** object with value 5

- one takes a String and returns an object

  Integer("45") returns an **`Integer`** object with value 45

  `Character` wrapper class is the only one which does not have this constructor

Wrapper Classes – Example (1)

- Prior to Java 5, the following statement caused a compiler error:

```
Float f = 234.567f;      // Error
```

  - Error text usually along the lines of: Cannot convert literal value to Float object, Incompatible data types
  - Solution was to explicitly use wrapper constructors or extraction methods such as intValue(), longValue()

- Similarly, the following sequence would not work:

```
float value = 234.567f;   // primitive
Float f = value;          // Error
```

Primitive values converted to objects

- Almost everything in Java exists in the form of an object
  - Except the 8 primitive data types
- Converting backwards and forwards between primitives and their object equivalents, can be tedious to program

- Solution ?
  - Autoboxing and Auto-Unboxing

Wrapper Classes – Example (1) revisited

- Since Java 5, the following statement is fine:

```
Float floatObject = 234.567f;
```

  - Java automatically converts the primitive float literal, into a Float object
  - This process is called **Autoboxing**

- Since Java 5, the following statement is also fine:

```
float value = floatObject;
```

  - Java automatically converts the Float object to a primitive float value to assign to the primitive variable
  - This process is called **Auto-Unboxing**

Wrapper Classes – Example 2

- If we want to have an ArrayList whose values are integers:
  - We cannot say `ArrayList<int> list` because int is not a class type
  - Instead we must use the Integer class:

```
ArrayList<Integer> list
```

  - We can use primitive int values to add:

```
list.add(6);
```

  - Autoboxing will convert the 6 (an int) to an equivalent Integer object.
  - We can retrieve and store into an int:

```
int myVal = list.get(4);   // Gets 5th item
```

Part 3:
Aggregation

Objectives – Part 3

- Write code to implement aggregation relationships using arrays or ArrayLists;
- Describe the role of and be able to implement a data management class;
- Explain how the concept of privacy leaks could arise when using arrays or ArrayList objects
- Write code so that it is not liable to privacy leaks in relation to arrays and ArrayLists;

Aggregation

- Aggregation is a relationship between classes where several instances of one class are gathered together in another class
- Differs from composition:
  - The instances exist and are created independent of the aggregate
  - The instances are not intrinsic to the aggregate
  - The instances can be simultaneously associated to other aggregates

Aggregation - Examples

- A group of students enrolled for FIT2034
- A group of students enrolled for FIT2003
  - Some students may be in both groups simultaneously
- A flock of sheep forms an aggregate
  - Some of the flock could be sold to become part of a different flock of a different farmer, the sheep are not dependent on the flock's existence in order to exist themselves
- The bank has an aggregation of BankAccounts and Customers.

Using Aggregates for Data Management

- A Data Management class is one designed with the specific responsibility to manage a collection of objects of a common type
  - An application of the concept of Aggregation
- Provides controlled-access methods to manipulate the collection
  - For example, you can search on particular criteria specific to the type of object you are collecting
- Usually uses either an array or ArrayList internally to manage the collection
  - It should not matter to users of the Data Management class, and you could change the implementation later.

Example Data Management Class

- A System for a Supermarket/Grocery may need a class to aggregate all the FoodType objects that represent all products sold
  - We will name it ProductsDatabase
- When the barcode is read, we need to find the matching FoodType object
  - We ask the ProductDatabase object to give us the FoodType object that has the corresponding bar code that we supply
- When new products become available to sell, we will need to add them to the ProductsDatabase

ProductsDatabase using an array (1)

```
public class ProductsDatabase {
  private FoodType[] products;
  private int totalProducts;

  public ProductsDatabase( ) {
     products = new FoodType [100];
     totalProducts = 0;
  }
  public boolean addProduct(FoodType newProduct) {
     boolean succeeded = false;
     if ((newProduct != null) &&
     (totalProducts < products.length) )  {
          products[totalProducts] = newProduct;
          totalProducts++;
          succeeded = true;
     }
     return succeeded;
  }
```

ProductsDatabase using an array (2)

```
public FoodType findProduct(int barcode) {
    boolean found = false;// assume the worst!
    int index = 0;

    while ( !found  &&  (index < totalProducts) )  {
        // examine current product:
        if ( products[index].getBarcode() == barcode)
            found = true;  // stop looking
        else
            index++;       // try next slot of array
    }
    if (found)
        return products[index];
    else
        return null;
}
```

if found is true, then index is the position of the product.

Privacy Leak Issues with Arrays/ArrayLists

- Sometimes we want to obtain a list of all the items in a Data Management class
  - So we could display them, for example
- However, the following implementation would give rise to a privacy leak:

```
public FoodType[ ] getAllProducts()
{
        return products;
}
```

- The internal array's reference is leaked to external caller method, bypassing privateness

Privacy leaks – Passing arrays to methods

- Array references are just object references
  - When you pass an array reference you are passing the array's address so the called method can have side effects
  - Actual and formal parameters are aliases
    - i.e. they both reference (point at) the same array in memory

```
public static void main(String[] args) {
   int[] marks ={10, 45, 67, 88, 90};

   for (int i = 0; i < marks.length; i++)
      System.out.print(marks[i] + " ");
   System.out.println();

   // syntax: formal parameter is just array name no []
   testArrayAsParameter(marks);

   for (int i = 0; i < marks.length; i++)
      System.out.print(marks[i] + " ");
}
private static void testArrayAsParameter(int[] passedArray){
   for (int i = 0; i < passedArray.length; i++)
      passedArray[i] = 2 * passedArray[i];
}
```

Note the syntax for actual and formal array parameters. No square brackets after actual parameters.

10 45 67 88 90
20 90 134 176 180

Arrays and Privacy Leaks – Cloning

- Array references are just object references
  - Therefore if they are instance variables mutators and accessors that pass these references in and out respectively will cause Privacy leaks
  - Instead independent copies of arrays must be passed in and out
- All arrays have a clone() method which when invoked on an array returns a copy of the array
  - We must typecast the clone to be an array of the right type,
    e.g. (FoodType[ ]) products.clone()
  - clone performs a “shallow copy”, i.e. if the array is an array of object references these objects are not copied, just their references.

Example – Full Demo Program

- On Moodle is a small demo about Sport Teams
- It demonstrates associations using arrays and ArrayList objects
- Establishes (bi-directional) links between Player and Team objects
- Uses an array of Teams (in the League class) for the system-wide set of known teams.
- Uses an ArrayList of Players to know who is in each team.