CMT205 Object Oriented Development with Java

WEEK 3

Methods, Arrays and Exceptions

METHODS

Methods

Definition

A set of statements to solve some subproblem should be coded as a ***method***.

```
// Example method to count sheep
public static int countSheep( int n ) {
    int sheepCount = n * n;
    return sheepCount;
}
```

static methods

- At first, the methods we write will be static.
- Static methods are methods that do not operate on objects.
- For example, the `pow` method of the `Math` class is a `static` method. The expression:

```
Math.pow(x,a);
```

- computes the power x^a, but you do not need to instantiate a Math object to use the method. i.e.

```
Math myMath = new Math();
myMath.pow(x,a);
```

NOT NEEDED

`static` method – general form:

```
public static returnType methodName( parameter_list ) {
    // Method body
}
```

static methods

Example – Method No Return [void]

```
public static void outputInt( String s, int n ) {
    System.out.println(s + " " + n);
}
```

Example – Method With Return [int]

```
public static int squareNumber( int n ) {
    int result = n * n;
    return result;
}
```

```
public class Foo {
    public static void main(...) {
        bar();
    }
    public static void bar() { ... }
}
```

```
public class Bob {
    public static void main(...) {
        Foo.bar();
    }
}
```

Parameter list (comma separated)

- Each parameter consists of a **type** and **identifier**.

Definition

- ***explicit parameters***: parameters in the method declaration.
- ***arguments***: parameters supplied when the method is called.

EXAMPLE: MethodScope.java

Remember -> Each parameter is only in scope in the corresponding method body.

```
public static void main( String[] args ) {
      squareNumber(5);
      System.out.println(result);
}


public static int squareNumber( int n ) {
      int result = n * n;
      return result;
}
```

OUT OF SCOPE

Adding Two Numbers

EXAMPLE: EasyAdd.java

```
// EasyAdd.java
// Adds two specified integers.
import java.util.Scanner;

public class EasyAdd {
   public static void main( String args[] ) {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter first number: ");
      int x = in.nextInt();
      System.out.print("Enter second number: ");
      int y = in.nextInt();
      int sum = x + y;
      System.out.println( "Their sum is " + sum);
   }
}
```

Application to add together 2 integers, requested from the user by using the Scanner class. **TASK:** Lets now re-write our application to create a method that will get integer values using the Scanner Class.

Adding Two Numbers

EXAMPLE: SimpleAddWithMethod.java

```
// SimpleAddWithMethod.java
// Adds two specified integers using a getInt()
method.
import java.util.Scanner;

public class SimpleAddWithMethod {
    public static void main( String args[] ) {
        int x = getInt("Enter first integer");
        int y = getInt("Enter second integer");
        int sum = x + y;
        System.out.println( "Their sum is " + sum);
    }

    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result;
    }
}
```

METHOD CALLS

METHOD

NOTE: Same functionality, but this time using a **method**

EXAMPLE: PrimitiveArguments.java

Passing arguments (primitive types)

Primitive type variables are **passed** using ***call by value*** – a **copy** of the variable is passed to the method.

```
public static void main( String[] args ) {
     int i = 0;
     System.out.println( "Before someMethod i = " + i );
     someMethod( i );
     System.out.println( "After someMethod i = " + i );
}

static void someMethod( int a ) {
     System.out.println( " In someMethod a = " + a );
     i++;
     System.out.println( " In someMethod a = " + a );
}
```

EXAMPLE: ReferenceArguments.java

Passing arguments (Objects)

- **Objects** are **never passed** as **arguments** to **methods**
- References to objects are passed using ***call by reference*** *value (i.e. copy of the object reference)*

```
public static void main( String[] args ) {
     java.awt.Point p = new java.awt.Point( 20, 20 );
     System.out.println( "Before someMethod p = " + p );
     someMethod( p );
     System.out.println( "After someMethod p = " + p );
     someOtherMethod( p );
     System.out.println( "After someOtherMethod p = " + p );
}
```

```
static void someMethod( java.awt.Point q ) {
     System.out.println( " On entering someMethod q = " + q );
     q.translate( 10, 10 );
     System.out.println( " On leaving someMethod q = " + q );
}
```

EXAMPLE: ReferenceArguments.java

Passing arguments (Objects)

```
public static void main( String[] args ) {
     java.awt.Point p = new java.awt.Point( 20, 20 );
     System.out.println( "Before someMethod p = " + p );
     someMethod( p );
     System.out.println( "After someMethod p = " + p );
     someOtherMethod( p );
     System.out.println( "After someOtherMethod p = " + p );
}
```

```
static void someOtherMethod( java.awt.Point q ) {
     System.out.println( " On entering someOtherMethod q = " + q );
     q = new java.awt.Point( 50, 50 );
     System.out.println( " In someOtherMethod q = " + q );
     q.translate( 10, 10 );
     System.out.println( " On leaving someOtherMethod q = " + q );
}
```

Group related methods

Lets go back to our adding application

```
public static void main( String args[] ) {
    int x = getInt("Enter first integer");
    int y = getInt("Enter second integer");
    int sum = x + y;
    System.out.println( "Their sum is " + sum);
}

public static int getInt( String prompt ) {
    Scanner in = new Scanner(System.in);
    System.out.print( prompt + " : " );
    int result = in.nextInt();
    return result;
}
```

- Why copy and paste this **method** into every class that needs it!!
- Lets create a ***class*** that can be easily reused.

Input Class

```
import java.util.Scanner;

public class Input {
    // Prompted input of an int
    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result;
    }
    // Prompted input of an float
    public static float getFloat( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        float result = in.nextFloat();
        return result;
    }
    // Prompted input of an word/token
    public static String getToken( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        String result = in.next();
        return result;
    }
}
```

Using our new class

Lets now write an application to make use of your new class:

```
public class SimplerAdd {
    public static void main( String args[] ) {
        int x = Input.getInt("Enter first integer");
        int y = Input.getInt("Enter second integer");
        int sum = x + y;
        System.out.println( "Their sum is " + sum);
    }
}
```

But the compiler and JVM must be able to find the **Input** class (i.e. the .class file)

(Note: it will automatically find files that are in the **same folder** as the **.java file**)

This can be inconvenient though, so we can use a **directory** instead and tell the compiler where to find the .class file.

Classpath

The ***classpath*** specifies a list of all **directories** and **jar** files that the JVM or Java compiler searches to **find class** files.

The classpath is set using an **extra argument** when executing the command `javac`:

```
javac -classpath .;"C:\io_examples" MyTestFile.java
```

The classpath also needs to be set (in a similar way) when using the Java virtual machine:

```
java -classpath .;"C:\io_examples" MyTestFile
```

An alternative is to use **Packages**

Packages

A **package** is a **library unit** that you get when you use the **import** keyword to bring in an entire library.

```
package mypackage; // Define package in class

public class MyClass {
    // Class definition
}
```

To use the class in programs either the **full name** must be specified:

```
mypackage.MyClass c = new mypackage.MyClass();
```

or the **package/class** must be **imported**:

```
import mypackage.*;
    MyClass c = new MyClass();

import mypackage.MyClass;
    MyClass c = new MyClass();
```

Packages

Locating files

- All class files are placed in a **single directory**
- To avoid name clashes, the **reverse** of the **author's domain name** should be used
- The directory structure matches the the package name

`static` vs `instance` methods

As outlined, a **`static`** method is special, you can call it without first creating an object of the class in which the method is declared.

The **`sqrt`** method in the class **`Math`** is **`static`**. This method does not need to be invoked on a **`Math`** object, instead we just write:

```
double d = Math.sqrt(25);
```

static method

The method **`translate`** in the **`Point`** class is **not static** – it **must** be invoked on a **`Point`** object

```
Point p = new Point(0,0);
p.translate(10,33);
```

instance method

Input class

Improving our `Input` class

- Each method call creates a new **`Scanner`** object
- **Adding** a **`field`** and a **`constructor`** to the class, we can avoid this and **remove** the Scanner from each method call:

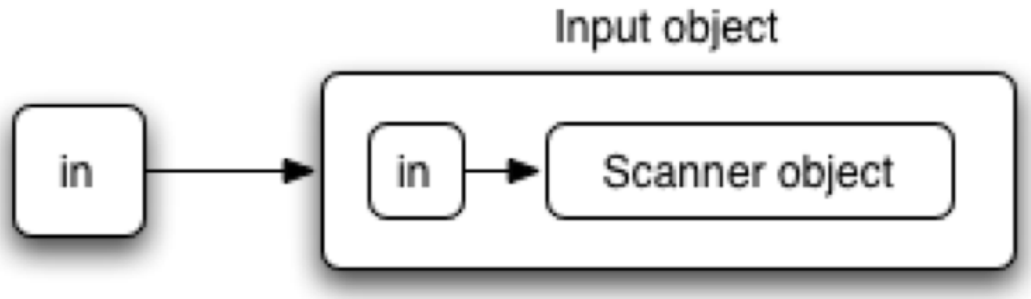
```
public class Input {
    Scanner in; // Instance Variable

    // Constructor
    public Input() {
        in = new Scanner(System.in);
    }

    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result; }
}
```

Remove **static** and becomes **instance** method

Input class

Improving our `Input` class

- The **methods** of the class are **no longer `static`**.
- Each object of our **`Input`** class is ***composed*** of a single **`Scanner`** object.



New Input class

EXAMPLE: Input.java

```
package uk.ac.cf.cs.examples.io;
import java.util.Scanner;

public class Input {
   Scanner in;

   // Constructor
   public Input() {
      in = new Scanner(System.in);
   }

   // Prompted input of an int
   public int getInt( String prompt ) {
      System.out.print( prompt + " : " );
      int result = in.nextInt();
      return result;
   }

   // Prompted input of a float
   public float getFloat( String prompt ) {
      System.out.print( prompt + " : " );
      float result = in.nextFloat();
      return result;
   }

   // Prompted input of a word/token
   public String getToken( String prompt ) {
      System.out.print( prompt + " : " );
      String result = in.next();
      return result;
   }
}
```

Using the new Input class

EXAMPLE: SimplerAdd.java

```
import uk.ac.cf.cs.examples.io.Input;

public class SimplerAdd {
     public static void main( String args[] ) {
          Input in = new Input();
          int x = in.getInt("Enter first integer");
          int y = in.getInt("Enter second integer");
          int sum = x + y;
          System.out.println( "Their sum is " + sum);
     }
}
```

Method overloading

```
public class Output {
  public static void printArray( int[] a ) {
     for ( int i = 0; i < a.length; ++i {
        System.out.print( a[i] );
        if ( i < a.length - 1 )
           System.out.print( ", " );
        }
        System.out.println(); }

  public static void printArray( String[] a )
{
     for ( int i = 0; i < a.length; ++i ) {
        System.out.print( a[i] );
        if ( i < a.length - 1 )
           System.out.print( ", " );
        }
        System.out.println();}
}
```

Method overloading

- Note that both methods on the previous page have **identical names** but **differ** in their **arguments**:

```
public static void printArray( int[] a )

public static void printArray( String[] a )
```

- A method is **overloaded** if the class containing it has at least **one other method** with the **same name** (but with different parameter types).
- Any number of methods can be defined with the same name as long as each overloaded method takes a **unique list of argument** types.

Worked example

Worked example: writing classes

Problem

- Create a phone book application that stores names and numbers, allows a user to add entries and retrieve the number for a given name, and load/save the data to file.

Classes

- PhoneBookEntry:
- PhoneBook:

Worked example

`PhoneBookEntry` Implementation

```
public class PhoneBookEntry {
    String name; // instance variables
    String number; // to store data
}
```

1. We want to ensure that every entry has a name and a number - **CONSTRUCTORS**
2. We'd like to control how code accesses the name and number – **ACCESS SPECIFIERS**
3. We'd like to make it easy to output `PhoneBookEntry` objects – **OVERRIDE toString()**

Worked example

Constructors (definition)

```
public class MyClass {
    public MyClass() {
        // Initialisation code here
    }
}
```

Remember:

A **constructor** is a special method that allows you to control how objects are instantiated.

Worked example

Constructors (PhoneBookEntry)

```
public class PhoneBookEntry {
   public PhoneBookEntry( String inName, String inNumber ) {
        name = inName;
        number = inNumber;
    }
}
```

OR

```
public class PhoneBookEntry {
   public PhoneBookEntry( String name, String number ) {
        this.name = name;
        this.number = number;
    }
}
```

`this` refers instance variables

Notes on constructors:

- Constructors **do not** have a **return value** specified.
- **Constructors** must have the **exact same name** as the **class**.
- If you create a constructor that takes arguments, then you cannot use a default constructor unless you explicitly define one.

Worked example

Access specifiers – `public`

The **`public`** keyword means that the following member or class declaration is available to everyone.

MyClass.java

```
public class MyClass {
    public void someMethod() {
        // Implementation
    }
}
```

MyOtherClass.java

```
public class MyOtherClass {
    public void someOtherMethod() {
        MyClass m = new MyClass();
        m.someMethod();
    }
}
```

Worked example

Access specifiers – `private`

The **`private`** keyword means that no one has access to the method except inside others of that particular class.

MyClass.java

```
public class MyClass {
    private int myField;
    private void someMethod() {
        // Implementation.
    }
    public void otherMethod() {
        System.out.prinltn(myField);
        someMethod();
    }
}
```

MyOtherClass.java

```
public class MyOtherClass {
    private void print() {
        MyClass m = new MyClass();
        m.someMethod();
        System.out.prinltn(myField);
    }
}
```

Worked example

Access specifiers – `protected` and package access

Definition (**`protected`**)

- The `protected` keyword means that access is only allowed from within the classes that extend the particular class.

Definition (**package access**)

- If no access specifier is included then the member is only available to other classes in the same package. This is sometimes termed as *friendly* access.

Worked example

Access specifiers – Accessors

private data

```
public class PhoneBookEntry {
    private String name;
    private String number;
}
```

Accessor

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

An ***accessor*** method allows ***read access*** to a private field (instance variable).

Worked example

Access specifiers – Mutators

private data

```
public class PhoneBookEntry {
    private String name;
    private String number;
}
```

Accessor

```
public String setName() {
    name = “matt”;
}
```

An ***mutator*** method allows ***write access*** to a private field (instance variable).

Worked example

String and StringBuffer classes

- As a quick aside, lets introduce the StringBuffer classes.
- String objects are *immutable* (i.e. they cannot be modified) whereas StringBuffer objects are *mutable* and can be modified.

String

```
String s = “abc”;
s = s.toUpperCase();
s.toLowerCase();
```

StringBuffer

```
StringBuffer sb = new StringBuffer(“ABC”);
sb.append(“DEF”);
```

Worked example

Easy output of the `PhoneBookEntry` class

- We want a way to output an entry to standard output.
- We can achieve this, but overriding the toString() method.
- We'll come back a little later ot look at exactly what is going on here.
- If me add the following method:

```
public String toString() {
    String s = name + "\t(" + number + ") ";
    return s;
}
```

- We can then print phonebook entries.

Worked example

EXAMPLE: PhoneBookEntry.java

Finished `PhoneBookEntry` application

```
class PhoneBookEntry {

    private String name;
    private String number;

    public PhoneBookEntry( String inName, String inNumber ) {
        name = inName;
        number = inNumber;
    }

    public String getName( ) {
        return name;
    }

    public String getNumber( ) {
        return number;
    }

    public void setNumber( String inNumber ) {
        number = inNumber;
    }

    public String toString( ) {
        String s = name + "\t(" + number + ") ";
        return s;
    }
}
```

Worked example

EXAMPLE: PhoneBookEntryTest.java

PhoneBookEntryTest application

Lets now write an application to use our new PhoneBookEntry class and test our code:

```
public class PhoneBookEntryTest {
    public static void main( String[] args ) {
        PhoneBookEntry e = new PhoneBookEntry( "Bob", "+44
          0123456789" );
        System.out.println( e ); }
    }
}
```

Anatomy of a Java class

EXAMPLE: JavaClass.java

```
public class JavaClass {

    private final String name;
    private String description;
    private static int staticInstanceVariable = 0;
    private int instanceVariable = 1;

    public JavaClass( String id, String description ){
            name = id;
            this.description = description;
    }

    public static void staticMethod() {
            System.out.println("JavaClass static method.");
    }

    public static int staticMethodWithReturn() {
            return staticInstanceVariable;
    }
    public void instanceMethod() {
            System.out.println("JavaClass instance method.");
    }

    public int instanceMethodWithReturn() {
            return instanceVariable;
    }

    public String toString() {
            return name + " -> " + description;
    }

    public static void main( String[] args ) {

            // Using static methods

            JavaClass.staticMethod();
            staticMethod();
            System.out.println( JavaClass.staticMethodWithReturn() );
            System.out.println( staticMethodWithReturn() );

            // Using instance methods - need to call from instantiated object

            JavaClass jc = new JavaClass("matts_class", "Anatomy of a Java Class");
            jc.instanceMethod();
            System.out.println( jc.instanceMethodWithReturn() );
            System.out.println( jc );

    }

}
```

class name

this refers to instance variable, without this it refers to local variable

instance variables

constructor

static methods

note static variable in static method

instance methods

override toString()

instantiate new JavaClass object

invoke constructor

use dot operator to access methods of JavaClass object

automatically invoke toString() method

ARRAYS

Arrays

- An ***array*** is a collection of data items of the same type.
- Every element of the collection can be accessed separately.
- An ***array*** with a given number of ***elements*** is constructed as

```
new typeName[length]
```

NOTE: **typeName** could be **int** (or other types) and **length** is the number of **elements** in the **array**

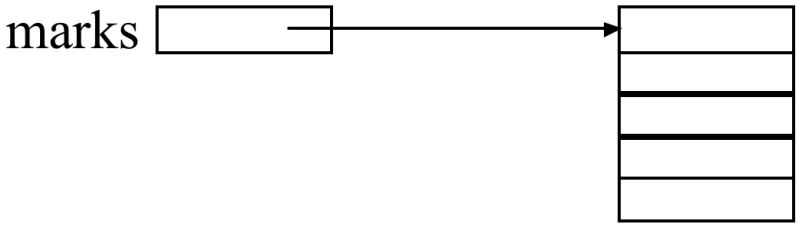
- An array has a fixed number of elements once created.

Array Example

To **create** an **array** which can store the **integer** marks of ***5*** students:

```
int[] marks = new int[5];
```

- marks is a **reference** to an **array** object
- the call to new int[5] creates the actual array of 5 integer numbers



Array Example

A **Java array** has an **instance** variable **length** which you can access to find out the **size** of an **array** called **marks**:

```
marks.length
```

When an **array** is first created, all elements are initialised to the default value

- Number (int, double etc.): **0**
- Boolean: **false**
- Object reference: **null**

Example

Mean.java (without using array)

```
// Program to calculate the mean mark
public class Mean
{
   public static void main( String[] args )
   {
      int mark1 = 56;
      int mark2 = 23;
      int mark3 = 45;
      int mark4 = 75;
      int mark5 = 61;
      int sum, mean;
      sum = mark1 + mark2 + mark3 + mark4 + mark5;
      mean = sum / 5;
      System.out.println( "Average mark of students is " + mean );
   }
}
```

Output:

```
Average mark of students is 52
```

Example

MeanArray.java (using array)

```
// Program to calculate the mean mark
public class MeanArray
{
   public static void main( String[] args )
   {
      int[] marks = new int[5];
      int sum, mean;
      marks[0] = 56;
      marks[1] = 23;
      marks[2] = 45;
      marks[3] = 75;
      marks[4] = 61;
      sum = marks[0] + marks[1] + marks[2] + marks[3] + marks[4];
      mean = sum / 5;
      System.out.println( "Average mark of students is " + mean );
   }
}
```

Output:

```
Average mark of students is 52
```

Example

MeanArrayInit.java (using array initialisation)

```
// Program to calculate the mean mark
public class MeanArrayInit
{
   public static void main( String[] args )
   {
      int[] marks = { 56, 23, 45, 75, 61 };
      int sum = 0;
      int mean;
      for ( int i = 0; i < 5; i++ )
      {
        sum += marks[i];
      }
      mean = sum / marks.length;
      System.out.println("Average mark of students is " + mean );
   }
}
```

Output:

```
Average mark of students is 52
```

Copying Arrays

- **Array** variables hold a **reference** to the actual **array**.
- If you copy the **reference**, you get another **reference** to the **same array**.
- If you want to make a true copy of an **array**, you must:
  - create a **new array** of the **same length** as the **original array**
  - copy over all the **values** from the **original array** to the **new array**
  - Alternatively, use static method:

```
System.arraycopy(srcObject, srcPos,
                 destObject, destPos, length)
```

- When two array objects are compared using e.g. ==, only the **references** are compared, **not** the contents.

Example

EXAMPLE: ArrayReference.java

ArrayReference.java

```
// Multiple references to an array
public class ArrayReference
{
   public static void main( String[] args )
   {
      int[] marks = { 56, 23, 45, 75, 61 };
      int[] examMarks;
      examMarks = marks;
      System.out.println("Mark of first student was "
                                          + marks[0] );
      marks[4] = 99;
      System.out.println("Mark of fifth student was "
                                            + examMarks[4] );
   }
}
```

Example

CompareReferences.java

```
// Comparison of array references
public class CompareReferences
{
   public static void main( String[] args )
   {
      int[] marks = { 56, 23, 45, 75, 61 };
      int[] assessmentMarks = marks;
      int[] examMarks = new int[marks.length];
      if ( marks == assessmentMarks )
         System.out.println("Same array");
      else
         System.out.println("Different arrays");
      if ( marks == examMarks )
         System.out.println("Same array");
      else
         System.out.println("Different arrays");
   }
}
```

Example

SystemArrayCopy.java

```
// Copying all elements from one array to another array
public class SystemArrayCopy
{
   public static void main( String[] args )
   {
      int[] marks = { 56, 23, 45, 75, 61 };
      int[] examMarks = new int[marks.length];


      System.arraycopy( marks, 0, examMarks, 0, marks.length);


      System.out.println("marks array:");
      for( int i = 0; i < marks.length; i++ ) {
          System.out.println( marks[i]);
      }


      System.out.println("examMarks array:");
      for( int i = 0; i < examMarks.length; i++ ) {
          System.out.println( marks[i]);
      }


      if ( marks == examMarks ) {
         System.out.println("Same array reference");
      }
      else {
         System.out.println("Different arrays");
      }
   }
}
```

Example

ArrayCopy.java

```
// Copying all elements from one array to another array
public class ArrayCopy
{
   public static void main( String[] args )
   {
      int element;
      int[] marks = { 56, 23, 45, 75, 61 };
      int[] examMarks = new int[marks.length];

      for ( element = 0; element < marks.length ; element++ ) {
         examMarks[ element ] = marks[ element ];
      }

      System.out.println("Mark of first student was " + marks[0] );
      System.out.println("Mark of fifth student was "
                                          + examMarks[4] );
   }
}
```

Example

ArrayReference.java

```
Mark of first student was 56
Mark of fifth student was 99
```

CompareReferences.java

```
Same array
Different arrays
```

ArrayCopy.java

```
Mark of first student was 56
Mark of fifth student was 61
```

SystemArrayCopy.java

```
Mark of first student was 56
Mark of fifth student was 61
```

Array Extension

- Arrays are of fixed lengths after creation
- To extend an array, you need to
  - create a new array with larger length
  - copy the content of the current array to the new array
  - assign the reference of the new array to the current
- This is expensive. To avoid frequent array extension, the new array is often larger than currently needed (e.g. twice the length)

Example

ArrayExtension.java

```
import java.util.Scanner;

public class ArrayExtension
{
   public static void main( String[] args )
   {
  Scanner in = new Scanner(System.in);
      final int ARRAY_SIZE = 5;
      int[] marks = new int[ ARRAY_SIZE ];
      int dataSize = 0;
      int studentMark;
      int sum=0;
      int mean =0;
      boolean done = false;

      // read marks until negative mark entered
      while ( ! done )
      {
         System.out.print( "Enter mark: " );
         studentMark = in.nextInt();
         if ( studentMark < 0 )
            done = true;
         // if more room in array, insert mark in array
         else if ( dataSize < marks.length )
    {
        marks[ dataSize ] = studentMark;
        sum += studentMark;
        dataSize++;
    }
```

Example [cont'd]

ArrayExtension.java

```
      else
         // create new array of double the size, copy
         // all elements from the old array to the new
         // array and insert mark into the new array
         {
        int[] newMarks = new int[ 2
                                          * marks.length ];
          System.arraycopy( marks, 0, newMarks, 0,
                                            marks.length );
        marks = newMarks;
        marks[ dataSize ] = studentMark;
        sum += studentMark;
        dataSize++;
         }
      }


      // output mean if at least one mark entered
      if ( dataSize != 0 )
      {
         mean = sum / dataSize;
         System.out.println( "Average mark is " + mean );
         System.out.println( "Array size is " + marks.length );
      }
      else
         System.out.println( "No marks entered" );
   }
}
```

Array Parameters

- When an ***array*** is passed to a ***method***, the ***array parameter*** contains a copy of the ***reference*** to the *arraya*.
- The ***method*** can therefore ***modify*** the contents of the ***array*** in the ***calling program*** (i.e. Call by Value)
- A ***method*** can also ***return*** an ***array***.

Example

Search.java

```
public class Search
{
   // method to calculate the number of passes
   public static int passes( int[] data )
   {
      if ( data.length == 0 ) return 0;
      int count = 0;
      for ( int i = 0; i < data.length; i++ )
      {
         if ( data[i] >= 50 )
            count++;
      }
      return count;
   }
   // main method
   public static void main( String[] args )
   {
      int[] marks = { 56, 23, 45, 75, 61 };
      int number = passes( marks );
      System.out.println("Number of students passing is " + number );
   }
}
```

Example

EXAMPLE: ArraySortMethod.java

ArraySortMethod.java

```
import java.util.Arrays;

public class ArraySortMethod
{
   // method to calculate the number of passes
   public static int[] sort( int[] data )
   {
  Arrays.sort(data);
      return data;
   }
   // main method
   public static void main( String[] args )
   {
      int[] marks = { 56, 23, 45, 75, 61 };

      System.out.println("BEFORE METHOD CALL");
      for ( int i = 0; i < marks.length; i++ ) {
      System.out.println(marks[i]);
      }

      // Call sort method and pass marks array as a parameter
      sort( marks );


      System.out.println("AFTER METHOD CALL");
      for ( int i = 0; i < marks.length; i++ ) {
      System.out.println(marks[i]);
      }

   }
}
```

Character Arrays

- You can define arrays of other type, e.g. char arrays
- **String.toCharArray()** returns a char array with characters corresponding to the String
- **String.valueOf()** obtains a String object from a char array
  - String.valueOf(charArray)
  - String.valueOf(charArray, int offset, int count)

Example

CharArrayToString.java

```
public class CharArrayToString
{
   public static void main( String[] args )
   {
      char[] charArray =
         { 'J', 'i', 'm', ' ', 'E', 'v', 'a', 'n', 's' };
      String fullName = String.valueOf( charArray );
      System.out.println( fullName );
      String surName = String.valueOf( charArray, 4, 5 );
      System.out.println( surName );
   }
}
```

Example

StringToCharArray.java

```
public class StringToCharArray
{
   public static void main( String[] args )
   {
      String name = ”Matt Morgan";
      char[] charArray = name.toCharArray();
      System.out.println(
          String.valueOf( charArray, 5, 6 ) );
   }
}
```

Higher Dimensional Arrays

- In Java, higher dimensional (e.g. 2-dimensional) arrays are treated as arrays of arrays.
- Create a 2D regular array (initialised to zero)

```
int[][] A = new int[2][3];
```

- Create a 2D regular array with initial values

```
int[][] A = {{1, 3, 5}, {2, 4, 6}};
```

- Or

```
int[][] A = new int[][] {{1, 3, 5}, {2, 4, 6}};
```

Higher Dimensional Arrays

- Higher dimensional (e.g. 2D) arrays are treated as arrays of arrays:
  - You can create irregular arrays (not a regular grid), e.g.

```
// manual creation of 2d array
int[][] arr = new int[2][];
arr[0] = new int[] {1, 5, 7, 9};
arr[1] = new int[] {2, 4};
```

- You can access part of the higher dimensional array as low dimensional arrays. If arr is a 2D array (array of array), then arr[0] is a 1D (normal) array.

ArrayList

EXAMPLE: ArrayTest.java

- If the size of the array cannot be predetermined, you can
  - use the array extension and dynamically re-allocate memory when needed, or
  - use ArrayList (**java.util.ArrayList**), which implements a dynamic array list
    - This is a generic type which can use elements of any object type, e.g. Integer (for int), Double for (double)
    - Use ArrayList<ElementType> to specify a specific ArrayList type
    - You can use add method to append element to the end of the list
    - You can use get method to obtain the element at a position

Other Data Structures

EXAMPLE: ArrayTest.java

- **java.util.Vector:** very similar to **ArrayList**, but synchronised, i.e. correct even if multiple threads access the object concurrently (more later)
- Both **Vector** and **ArrayList** rely on arrays, so require memory reallocation when the current space runs out
- Efficient for adding/removing elements at the end, slow for inserting/removing elements in the middle
- **java.util.LinkedList** is an implementation for linked list, so suitable for dynamic update
- We will discuss about generics and Java collections in more detail later.

EXCEPTION HANDLING

Exception Handling

- Exception handling
  - is designed for dealing with ***synchronous errors*** (e.g. an attempt to divide by zero that occurs as the program executes the divide instruction)
  - is not designed to deal with ***asynchronous events*** (e.g. disk I/O completions and mouse clicks) --- these are best handled through ***event handlers.***
- Exception handling is used in error situations to allow the system to recover from a malfunction causing the ***exception.***
- The programmer must write the recovery procedure, which is called an ***exception handler.***

Exception Handling

- Common examples of ***exceptions*** are:
  - *Array subscript out of bounds*
  - *Division by zero*
  - *Memory exhaustion*

- You can also define your own kinds of exceptions for issues arising in your own methods.

Setting up an Exception Handler

- The code that may generate an ***exception*** is enclosed in a ***try*** block.
- The ***try*** block is immediately followed by one or more ***catch*** blocks.
- Each ***catch*** block specifies the type of ***exception*** it can ***catch*** and contains program code which is known as an ***exception*** handler.
- After the last ***catch*** block, an optional ***finally*** block provides code that is always executed regardless of whether or not an ***exception*** occurs.

Format of an Exception Handler

General form

```
try {
  // Code that may throw exception
}
catch (ExceptionType e) {
  // Handle exception
}
finally { // Optional
  // Tidy up
}
```

Example (Throwing exceptions)

```
if (Boolean-expression) {
  throw new NumberFormatException("any extra information");
}
```

Exceptions are objects of the Exception class (or a class that extends the Exception class). Useful methods defined in this class include:

- toString
- printStackTrace

Execution Process

- If a ***try*** block executes and no ***exceptions*** are thrown
  - all the exception handlers (i.e the ***catch*** blocks) are skipped
  - execution continues with the first statement after the last exception handler.
- If an exception is thrown
  - control exits the current ***try*** block
  - proceeds to the appropriate ***exception*** handler (first matched exception handler) (if any)
  - The statements in this block are executed and then execution continues with the first statement after the last ***exception*** handler

Execution Process [cont'd]

- If the ***finally*** block exists
  - It is always executed regardless of whether or not an ***exception*** occurs.
  - It often contains code that releases ***resources***.
  - If there are no ***catch*** blocks following a ***try*** block, the ***finally*** block ***must*** be provided.
- If an ***exception*** occurs which is not caught by one of the ***exception*** handlers
  - If a ***finally*** block is present, the ***finally*** block is executed
  - the ***exception*** is passed to a handler at some level above (returned to the caller)

Example

ExceptionTest.java

```
public class ExceptionTest {

  public static void main( String[] args ) {

    try {
      System.out.println(
             "Trying Integer.parseInt( \"NOT AN INT\" )"
);
      int i = Integer.parseInt( "Not an int" );
    }
    catch ( Exception e ) {
      System.out.println("Some error" + e);
      e.printStackTrace();
    }
    finally {
      System.out.println( "Finally always reached" );
    }
  }
}
```

Stack Traces

EXAMPLE: StackTrace.java

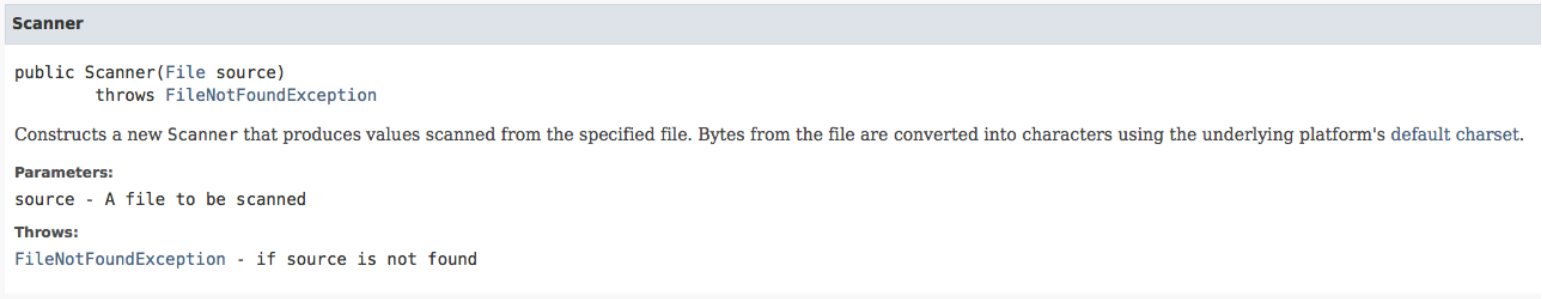
Definition

The Java virtual machine uses a ***call stack*** to track the currently executing method and the method that called the current method. In any Java application the ***main*** method is always the first method on the stack and the last method to leave the stack.

- When an exception is thrown, execution of the current method stops. The JVM looks for a handler in the current method
- If no handler is found, the exception is passed through the call stack until a handler is found
- The application will exit if no handler is found and information including a ***stack trace*** is printed

Checked Exceptions

For certain types of exception, called checked exceptions, Java insists that if a method can throw an exception, you **must** handle it.



Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:
source - A file to be scanned

Throws:
FileNotFoundException - if source is not found

Checked exceptions can be *handled* by:

- Use of try and catch blocks
- Denoting that your method may throw a checked exception:

```
public void myMethod(File f) throws FileNotFoundException {
    Scanner s = new Scanner(f);
    // Other code
}
```

EXAMPLE: Reverse.java

Example 1 – Read and Reverse

```
import java.util.Scanner;
import java.io.File;

public class Reverse {

  public static void main( String[] args ) {

    try {

      Scanner in = new Scanner( new File(args[0]) );

      while ( in.hasNextLine() ) {
        System.out.println( new StringBuffer(

        in.nextLine()).reverse() );
      }

      in.close();
    }
    catch ( Exception e ) {
      System.out.println( e );
    }
  }
}
```

Sum Doubles

EXAMPLE: Sum.java

Example 2 – Sum Doubles

```
import java.util.Scanner;
import java.io.File;

public class Sum {

  public static void main( String[] args ) {

    try {
      Scanner in = new Scanner( new File(args[0]) );

      double total = 0;
      while (in.hasNextDouble()) {
        total += in.nextDouble();
      }

      System.out.printf( "Sum is %.4f%n", total );

      in.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Feedback

Are there things you like?, Things you don't like?, Suggestions of any kind? I really would like to hear from you.

Let me know anonymously at :
https://www.suggestionox.com/r/It5Z11