# 10 Creating your own Classes

# Objectives

At the end of the lesson, the student should be able to:

- Create their own classes
- Declare properties (fields) and methods for their classes
- Use the this reference to access instance data
- Create and call overloaded methods
- Import and create packages
- Use access modifiers to control access to class members

# Defining Your Own Class

# Defining your own classes

- Things to take note of for the syntax defined in this section:

  * means that there may be 0 or more occurrences of the line where it was applied to.

  <description> indicates that you have to substitute an actual value for this part instead of typing it as it is.

  [ ] indicates that this part is optional

# Defining your own classes

- To define a class, we write:

```
<modifier> class <name> {
        <attributeDeclaration>*
        <constructorDeclaration>*
        <methodDeclaration>*
  }
```

  - where
    - <modifier> is an access modifier, which may be combined with other types of modifier.

# Example

```
public class StudentRecord {
    //we'll add more code here later
}
```

- where,
  - public - means that our class is accessible to other classes outside the package
  - class - this is the keyword used to create a class in Java
  - StudentRecord - a unique identifier that describes our class

# Coding Guidelines

- Think of an appropriate name for your class. Don't just call your class XYZ or any random names you can think of.
- Class names starts with a CAPITAL letter - not a requirement, however.
- The filename of your class must have the SAME NAME as your class name.

# Instance Variables

# Declaring Properties (Attributes)

- To declare a certain attribute for our class, we write,

```
<modifier> <type> <name> [=
  <default_value>];
```


JEDI

# Coding Guidelines

- Declare all your instance variables right after “public class Myclass {“
- Declare one variable for each line.
- Instance variables, like any other variables should start with a SMALL letter.
- Use an appropriate data type for each variable you declare.
- Declare instance variables as private so that only class methods can access them directly.
  - Encaptulation

# Static Variables

# Class (static) variables

```
public class StudentRecord {
    //static variables we have declared
    private static int studentCount;
    //we'll add more code here later
}
```

- we use the keyword static to indicate that a variable is a static variable.

# Methods

# Declaring Methods

- To declare methods we write,

```
<modifier> <returnType> <name>
(<parameter>*) {
        <statement>*
}
```

  - where,
    - <modifier> can carry a number of different modifiers
    - <returnType> can be any data type (including void)
    - <name> can be any valid identifier
    - <parameter> ::= <parameter_type> <parameter_name>[,]

# Accessor (Getter) Methods

- Accessor methods
  - used to read values from our class variables (instance/static).
  - usually written as:

    ```
    get<NameOfInstanceVariable>
    ```

  - It also returns a value.

# Example 1: Accessor (Getter) Method

```
public class StudentRecord {
    private String name;
    :
    public String getName(){
       return name;
    }
}
```

- where,
  - public - means that the method can be called from objects outside the class
  - String - is the return type of the method. This means that the method should return a value of type String
  - getName - the name of the method
  - () - this means that our method does not have any parameters

# Example 2: Accessor (Getter) Method

```
public class StudentRecord {
    private String name; //
    some code

    // An example in which the business logic is //
    used to return a value on an accessor method
    public double getAverage(){
        double result = 0;
        result=(mathGrade+englishGrade+scienceGrade)/3;
        return result;
    }
}
```

# Mutator (Setter) Methods

- Mutator Methods
  - used to write or change values of our class variables (instance/static).
  - Usually written as:

```
set<NameOfInstanceVariable>
```

# Example: Mutator (Setter) Method

```
public class StudentRecord {
    private String name;
    :
    public void setName( String temp ){
       name = temp;
    }
}
```

- where,
  - public - means that the method can be called from objects outside the class
  - void - means that the method does not return any value
  - setName - the name of the method
  - (String temp) - parameter that will be used inside our method

# Multiple return statements

- You can have multiple return statements for a method as long as they are not on the same block.
- You can also use constants to return values instead of variables.

# Example: Multiple return statements

```
public String getNumberInWords( int num ){
    String defaultNum = "zero";
    if( num == 1 ){
       return "one"; //return a constant
    }
    else if( num == 2){
       return "two"; //return a constant
    }
    //return a variable
    return defaultNum;
}
```

# Static Methods


JEDI

# Coding Guidelines

- Method names should start with a SMALL letter.
- Method names should be verbs
- Always provide documentation before the declaration of the method. You can use javadocs style for this. Please see example.

# Example Code

# Source Code for StudentRecord class

```
public class StudentRecord {

    // Instance variables
     private String    name;
     private String    address;
     private int    age;
     private double mathGrade;
     private double englishGrade;
     private double scienceGrade;
     private double average;
     private static int studentCount;
```

# Source Code for StudentRecord Class

```
/**
 * Returns the name of the student (Accessor method)
 */
public String getName(){
    return name;
}

/**
 * Changes the name of the student (Mutator method)
 */
public void setName( String temp ){
    name = temp;
}
```

# Source Code for StudentRecord Class

```
/**
 * Computes the average of the english, math and science *
 grades (Accessor method)
 */
public double getAverage(){
    double result = 0;
    result  =  (  mathGrade+englishGrade+scienceGrade  )/3;
    return result;
}
/**
 * returns the number of instances of StudentRecords *
 (Accessor method)
 */
public static int getStudentCount(){
    return studentCount;
}
```

# Sample Source Code that uses StudentRecord Class

```
public class StudentRecordExample
{
   public static void main( String[] args ){

    //create three objects for Student record
    StudentRecord annaRecord = new StudentRecord();
    StudentRecord beahRecord = new StudentRecord();
    StudentRecord crisRecord = new StudentRecord();

    //set the name of the students
    annaRecord.setName("Anna");
    beahRecord.setName("Beah");
    crisRecord.setName("Cris");

    //print anna's name
    System.out.println( annaRecord.getName() );

    //print number of students
    System.out.println("Count="+StudentRecord.getStudentCount());
   }
}
```

# Program Output

```
Anna
Student Count = 0
```

# “this” Reference

# “this” reference

- The this reference
  - refers to current object instance itself
  - used to access the instance variables shadowed by the parameters.

- To use the this reference, we type,

```
this.<nameOfTheInstanceVariable>
```

- You can only use the this reference for instance variables and NOT static or class variables.

# Example

```
public void setAge( int age ){
    this.age = age;
}
```

# Overloading Methods


JEDI

# Example

```
public void print( String temp ){
   System.out.println("Name:" + name);
   System.out.println("Address:" + address);
   System.out.println("Age:" + age);
 }

public void print(double eGrade, double mGrade,
                  double sGrade)
   System.out.println("Name:" + name);
   System.out.println("Math Grade:" + mGrade);
   System.out.println("English Grade:" + eGrade);
   System.out.println("Science Grade:" + sGrade);
 }
```

# Example

```
public static void main( String[] args )
{
    StudentRecord annaRecord = new StudentRecord();

    annaRecord.setName("Anna");
    annaRecord.setAddress("Philippines");
    annaRecord.setAge(15);
    annaRecord.setMathGrade(80);
    annaRecord.setEnglishGrade(95.5);
    annaRecord.setScienceGrade(100);

    //overloaded methods
    annaRecord.print( annaRecord.getName() );
    annaRecord.print( annaRecord.getEnglishGrade(),
              annaRecord.getMathGrade(),
                  annaRecord.getScienceGrade());
}
```

# Output

- we will have the output for the first call to print,

```
Name:Anna
Address:Philippines
Age:15
```

- we will have the output for the second call to print,

```
Name:Anna
Math Grade:80.0
English Grade:95.5
Science Grade:100.0
```

# Constructors
# (Constructor Methods)

# Constructors

- Constructors are important in instantiating an object. It is a method where all the initializations are placed.

- The following are the properties of a constructor:
  - Constructors have the same name as the class
  - A constructor is just like an ordinary method, however only the following information can be placed in the header of the constructor,
  - scope or accessibility identifier (like public...), constructor's name and parameters if it has any.
  - Constructors does not have any return value
  - You cannot call a constructor directly, it can only be called by using the new operator during class instantiation.

# Constructors

- To declare a constructor, we write,

```
<modifier> <className> (<parameter>*) {
                                <statement>*
}
```

# Default Constructor (Method)

- The default constructor (no-arg constructor)
  - is the constructor without any parameters.
  - If the class does not specify any constructors, then an implicit default constructor is created.

# Example: Default Constructor Method of StudentRecord Class

```
public StudentRecord()
{
    //some code here
}
```


JEDI

# Using Constructors

- To use these constructors, we have the following code,

```
public static void main( String[] args ){
    //create three objects for Student record
    StudentRecord annaRecord=new StudentRecord("Anna");
    StudentRecord beahRecord=new StudentRecord("Beah",
                        "Philippines");
    StudentRecord crisRecord=new StudentRecord
    (80,90,100);
    //some code here
}
```

# “this()” constructor call

- Constructor calls can be chained, meaning, you can call another constructor from inside another constructor.
- We use the this() call for this
- There are a few things to remember when using the this constructor call:
  - When using the this constructor call, IT MUST OCCUR AS THE FIRST STATEMENT in a constructor
  - It can ONLY BE USED IN A CONSTRUCTOR DEFINITION. The this call can then be followed by any other relevant statements.

# Example

```
1: public StudentRecord(){
2:     this("some string");
3:
4: }
5:
6: public StudentRecord(String temp){
7:     this.name = temp;
8: }
9:
10:  public  static  void  main(  String[]  args  )
11: {
12:
13:    StudentRecord annaRecord = new StudentRecord();
14: }
```

# Packages

# Packages

- Packages
  - are Java's means of grouping related classes and interfaces together in a single unit (interfaces will be discussed later).
  - This powerful feature provides for a convenient mechanism for managing a large group of classes and interfaces while avoiding potential naming conflicts.

# Importing Packages

- To be able to use classes outside of the package you are currently working in, you need to import the package of those classes.
- By default, all your Java programs import the java.lang.* package, that is why you can use classes like String and Integers inside the program even though you haven't imported any packages.
- The syntax for importing packages is as follows:

```
import <nameOfPackage>;
```

# Example: Importing Packages or Class

```
import java.awt.Color;
import java.awt.*;
```

# Placing a Class in a Package

- To place a class in a package, we write the following as the first line of the code (except comments)

```
package <packageName>;
```

```
package myownpackage;
```

- Packages can also be nested. In this case, the Java interpreter expects the directory structure containing the executable classes to match the package hierarchy.

```
package myowndir.myownsubdir.myownpackage;
```

# Example: Placing StudentRecord class in SchoolClasses pacakge

```
package SchoolClasses;

public class StudentRecord {
    private String name;
    private String address;
    private int      age;
    :
```

# Classpath

# Setting the CLASSPATH

- Now, suppose we place the package schoolClasses under the C:\ directory.

- We need to set the classpath to point to that directory so that when we try to run it, the JVM will be able to see where our classes are stored.

- Before we discuss how to set the classpath, let us take a look at an example on what will happen if we don't set the classpath.


JEDI

# Setting the CLASSPATH

- To set the classpath in Windows, we type this at the command prompt,

```
C:\schoolClasses> set classpath=C:\
```

  - assuming C:\ is the directory in which we have placed the packages meaning there is a directory C:\schoolClasses and there is a C:\schoolClasses\StudentRecord.class

- After setting the classpath, we can now run our program anywhere by typing,

```
C:\schoolClasses> java
schoolClasses.StudentRecord
```

# Setting the CLASSPATH

- For Unix base systems, suppose we have our classes in the directory /usr/local/myClasses, we write,

```
export classpath=/usr/local/myClasses
```

# Setting the CLASSPATH

- Take note that you can set the classpath anywhere. You can also set more than one classpath, we just have to separate them by ;(for windows) and : (for Unix based systems). For example,

```
set classpath=C:\myClasses;D:\;E:\MyPrograms\Java
```

- and for Unix based systems,

```
export classpath=/usr/local/java:/usr/myClasses
```

# Access Modifiers

# Access Modifiers

- There are four different types of member access modifiers in Java:
  - public
  - private
  - protected
  - Default
- The first three access modifiers are explicitly written in the code to indicate the access type, for the fourth one which is default, no keyword is used.

# default accessibility

- Default access
  - specifies that only classes in the same package can have access to the class' variables and methods
  - no actual keyword for the default modifier; it is applied in the absence of an access modifier.

# Example

```
public class StudentRecord {
    //default access to instance variable
    int name;

    //default  access  to  method
    String getName(){
       return name;
    }
}
```

# public accessibility

- public access
  - specifies that class members (variables or methods) are accessible to anyone, both inside and outside the class and outside of the package.
  - Any object that interacts with the class can have access to the public members of the class.
  - Keyword: public

# Example: “public” Access Modifer

```
public class StudentRecord {
    //default  access  to  instance  variable
    public int name;

    //default access to method
    public String getName(){
       return name;
    }
}
```

# protected accessibility

- protected access
  - specifies that the class members are accessible only to methods in that class and the subclasses of the class.
  - Keyword: protected

# Example: “protected” Access Modifier

```
public class StudentRecord {
    //default  access  to  instance  variable
    protected int name;

    //default access to method
    protected String getName(){
       return name;
    }
}
```

# private accessibility

- private accessibility
  - specifies that the class members are only accessible by the class they are defined in.
  - Keyword: private

# Example: “private” Access Modifier

```
public class StudentRecord {
    //default  access  to  instance  variable
    private int name;

    //default access to method
    private String getName(){
       return name;
    }
}
```

# Coding Guidelines

- The instance variables of a class should normally be declared private, and the class will just provide accessor and mutator methods to these variables.

Summary

| Access Levels | | | | |
|---|---|---|---|---|
| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| `public` | Y | Y | Y | Y |
| `protected` | Y | Y | Y | N |
| *no modifier* | Y | Y | N | N |
| `private` | Y | N | N | N |

# Summary

- Defining your own classes
- Declaring Fields (instance, static/class)
- Declaring Methods (accessor, mutator, static)
- Returning values and Multiple return statements
- The this reference
- Overloading Methods
- Constructors (default, overloading, this() call)
- Packages
- Access Modifiers (default, public, private, protected)