Murdoch
UNIVERSITY

# Topic 5 Programming with Classes

ICT167 Principles of Computer Science

Murdoch UNIVERSITY

## Objectives

- Explain what is meant by a **constructor**
- Explain what constructors are used for
- Know when Java will supply an **automatic constructor**
- Know when Java will not supply an automatic constructor
- Know how a client uses a class with an automatic constructor
- Define the term **default constructor**

## Objectives

- Explain why a class may have several constructors
- Be able to **supply several useful constructors** with a class
- Know **how to use the different constructors** which belong to a class
- Know when a set method is to be used rather than a constructor
- Define **overloading**

## Objectives

- Use overloaded methods in your own classes
- Know how overloading interacts with automatic type conversion
- Understand the dangers of passing objects as parameters (privacy leak !)
- Understand the enumeration (**enum**) type
- Be able to use enumerations in Java
- Be able to **import** library classes if necessary
- Define the term **package** in java

**Reading –** Savitch: Chapters 6.1, 6.4-6.7

## Constructors

- A constructor is a **special method** designed to initialize instance variables
- Automatically called when an object is created (by a client) using ***new***
- Given exactly the same name as the name of the class
- Can have parameters
- Cannot return a value, so has no return type, not even void

## Constructors

- Often there is more than one constructor for the same class definition
  - Different versions to initialize all, some, or none of the instance variables
  - Each constructor has a different signature (a different number or sequence of argument types)
- A constructor with no parameters is called a ***default constructor***

## Constructors

- If no constructor is provided, Java automatically creates a default constructor
- If *any* constructor is provided, then *no* constructors are created automatically
- Eg:

```
SpeciesFourthTry speciesOfTheMonth = new
                             SpeciesFourthTry();
```

- The `new` operator says to create a new object
  - It is followed by the name of a *constructor*

## Constructors

- `SpeciesFourthTry()` is the constructor in the above example
- Until now we have mostly been using automatic constructors which Java supplies for our classes (as in the above example)
- To replace the automatic constructor, the creator of the class can supply one or more of their own constructors inside the class definition

Murdoch UNIVERSITY

## Constructors

- Eg: the following statement, uses the constructor `String(String value)` which is part of the definition for the class String

```
// create an object of type String
String str=new String("Information Technology");
// create an object of type String - automatic
constructor
// String str=new String();
```

  - If the creator of the class supplies no constructors then Java will supply an automatic constructor
  - If the creator of the class supplies any constructor methods inside the class definition, then Java will not supply any constructors at all

Murdoch UNIVERSITY

## Constructors

- Constructors are mostly used to give initial values to instance variables of the new object
  - However, they may be useful to do some setting up of the screen for a GUI, or an external file, or a network connection associated with the object
- A class may have several constructors
  - They may differ in how they set up a new object
- By supplying several constructors, the creator is giving the client a choice of ways that new objects are set up

Murdoch UNIVERSITY

# Pet Class UML Class Diagram

| Pet |
|---|
| - name: String<br>- age: int<br>- weight: double |
| + setPet(String newName,int newAge,double newWeight): void<br>+ setName(String newName): void<br>+ setAge(int newAge): void<br>+ setWeight(double newWeight): void<br>+ getName(): String<br>+ getAge(): int<br>+ getWeight: double<br>+ writeOutput(): void |

## Example: Pet Class

```
/** Class for basic pet data: name, age, weight */
public class Pet {
  private String name;
  private int age;         //in years
  private double weight;//in pounds
  // default constructor
  public Pet() {
     name = "No name yet.";
     age = 0;
     weight = 0;
  }
```

Murdoch UNIVERSITY

## Example: Pet Class

```
// a constructor with three arguments, only
// called when you create an object with new
public Pet(String initialName, int initialAge,
                            double initialWeight) {
   name = initialName;
   if ((initialAge <0)||(initialWeight <0)) {
      System.out.println("Error: -ve age or weight.");
      System.exit(0);
   }else{
      age = initialAge;
      weight = initialWeight;
   }
} // end constructor
```

Murdoch UNIVERSITY

## Example: Pet Class

```
// a set method-to change an already existing object
public void setPet(String initialName, int
               initialAge, double initialWeight) {
   name = initialName;
   if ((initialAge <0)||(initialWeight <0)) {
      System.out.println("Error: -ve age or weight.");
      System.exit(0);
   }else{
      age = initialAge;
      weight = initialWeight;
   }
} // end
```

## Example: Pet Class

```
// another constructor
public Pet(String initialName) {
   name = initialName;
   age = 0;
   weight = 0;
}
// second set method
public void setName(String newName) {
   name = newName; //age, weight are unchanged
}
```

# Example: Pet Class

```
// another constructor
public Pet(int initialAge) {
   name = "No name yet.";
   weight = 0;
   if (initialAge < 0) {
      System.out.println("Error: -ve age.");
      System.exit(0);
    }else{
      age = initialAge;
    }
}
```

## Example: Pet Class

```
// third set method
public void setAge(int newAge) {
   if (newAge < 0) {
      System.out.println("Error: Negative age.");
      System.exit(0);
   }else{
      age = newAge;
       //name and weight are unchanged
   }
 }
```

## Example: Pet Class

```
// another constructor
public Pet(double initialWeight) {
   name = "No name yet";
   age = 0;
   weight = initialWeight;
}
// fourth set method
public void setWeight(double newWeight) {
   weight = newWeight;
   //name and age are unchanged.
}
```

## Example: Pet Class

```
// get methods
public String getName() {
   return name;
}
public int getAge() {
   return age;
}
public double getWeight() {
   return weight;
}
```

# Example: Pet Class

```
  public void writeOutput()
  {
     System.out.println("Name: " + name);
     System.out.println("Age: " + age + " years.");
     System.out.println("Weight: "+weight+" pounds.");
  }
} // end class Pet
```

Murdoch UNIVERSITY

# Things To Note About Pet Class

- There are several constructors but they differ in number or type of parameters. These are:

```
public Pet() {…}
public Pet(String initialName, int
  initialAge, double initialWeight)
{…}
public Pet(String initialName) {…}
public Pet(int initialAge) {…}
public Pet(double initialWeight) {…}
```

# Things To Note About Pet Class

- There is a DEFAULT constructor
  - That is, one with no parameters, and the creator is writing this themselves because Java will not supply any AUTOMATIC constructors for this class
- Despite there being constructors, we still need to supply set (mutator) methods in case the client wants to modify the values of the instance variables of already existing objects
- Below is an example client program of the Pet class

## Example: PetDemo Class

```
import java.util.Scanner;
public class PetDemo {
  public static void main(String[] args) {
     Pet yourPet = new Pet("Jane Doe");
     System.out.println("My records on your pet
                             are inaccurate.");
     System.out.println("Here is what they
                              currently say:");
     yourPet.writeOutput();
     System.out.println("Please enter the
                             correct pet name);
```

## Example: PetDemo Class

```
Scanner keyboard = new Scanner(System.in);
String correctName = keyboard.nextLine();
System.out.println("Please enter the
                           correct pet age:");
int correctAge = keyboard.nextInt();
System.out.println("Please enter the
                        correct pet weight:");
double correctWeight=keyboard.nextDouble();
yourPet.setPet(correctName, correctAge,
                                correctWeight);
```

## Example: PetDemo Class

```
        System.out.println("Updated records say:");
        yourPet.writeOutput();
    }
}// end class PetDemo
/* OUTPUT:
My records on your pet are inaccurate.
Here is what they currently say:
Name: Jane Doe
Age: 0 years
Weight: 0.0 pounds
```

## Example: PetDemo Class

```
Please enter the correct pet name:
Oscar
Please enter the correct pet age:
7
Please enter the correct pet weight:
15
My updated records now say:
Name: Oscar
Age: 7 years
Weight: 15.0 pounds
*/
```

## Things To Note About PetDemo

- The client invokes a constructor by using ***new***
- Which constructor is used depends on the arguments in the parentheses after the class name after ***new***
- When the client wants to change the data (i.e. instance variables) belonging to an existing object then they **do not** use a constructor

# Things To Note About PetDemo

- In general, if the creator supplies no constructors then Java will automatically supply one default constructor (i.e. one with no arguments) which gives all the instance variables certain initial values (like 0 or the null reference) depending on their types
- Any default constructor is also called via ***new***

```
MyClass m1 = new MyClass();
Pet yourPet = new Pet();
```

Murdoch UNIVERSITY

## Things To Note About PetDemo

- A constructor can call other methods in its class
  - Eg: constructors in the class Pet can be revised to call one of the set methods as follows:

```
public Pet(String initialName, int
  initialAge, double initialWeight) {
// call to class set method from constructor
setPet(initialName, initialAge, initialWeight);
}
```

## In Summary

- Always use a constructor after **new** when creating objects
- For example, using the Pet class above:

```
Pet myCat = new  Pet ("Kitty", 3,
6.5);
```

- This calls the `Pet` constructor with `String, int, and double` parameters
- If you want to change values of instance variables after you have created an object, you must use other set methods for the object

## In Summary

- You cannot call a constructor for an object after it is created
- Set methods should be provided for this purpose
- Calling class's **public methods** from its constructor can lead to problems particularly when using inheritance (see later) because it is possible for another class to alter the behaviour of the public method and thus adversely affect the behaviour of the constructor

## Overloading

- **Overloading** means using the same name for two or more methods within the same class
- We have already seen the use of methods with the same name in different classes (eg: `equals` in many classes, `charAt` in `String` and `StringBuffer` classes)
- It is also convenient to use the same name for closely related methods within ONE class

Murdoch UNIVERSITY

## Overloading

- The compiler must be able to tell which method is to be used in a particular call
  - So the *method signature* (i.e. number and/or types of parameters) must be different
- You may have already used overloaded methods in the Math class:

```
Math.max(2,3) returns the integer 3,
but
```

```
Math.max(2.5,6.5) returns the double
6.5
```

## Overloading

- The division operator is also overloaded to perform both integer and floating point operations (even though it is not exactly a method) :

  `3/2 evaluates to 1, but`

  `3.0/2.0 evaluates to 1.5`

- The methods `print` and `println` of Java library class `PrintStream` are also overloaded – each method takes one parameter which can be of type `String`, `int`, or `double`, etc.

## Example: Overload Class

```
/** This is just a toy class to illustrate
   overloading
 The class Overload has 3 different methods - all
 named getAverage() */
public class Overload {
  public static void main(String[] args) {
   double avg1 = Overload.getAverage(40.0,50.0);
   double avg2 = Overload.getAverage(1.0,2.0,3.0);
   char avg3 = Overload.getAverage('a','c');
   System.out.println("average1 = " + avg1);
   System.out.println("average2 = " + avg2);
   System.out.println("average3 = " + avg3);
  } // end main
```

Murdoch UNIVERSITY

## Example: Overload Class

```
  public static double getAverage(double first,
                                              double second) {
     return ((first + second)/2.0);
  }
  public static double getAverage(double first,
                           double second, double third) {
     return ((first + second + third)/3.0);
  }
  public static char getAverage(char first,
                                                char second) {
     return(char)(((int)first + (int)second)/2);
  }
}// end class Overload
```

## Example: Output

```
/* Output:
average1 = 45.0
average2 = 2.0
average3 = b
*/
```

## Things To Note

- In the above example, method `getAverage()` has 3 definitions within the same class
- Therefore, the method `getAverage()` **is overloaded**
- **Note**: each definition in this case must have a different **signature**. That is:
  - a different number of arguments/parameters, or
  - corresponding arguments/parameters must have different types

## Things To Note

- A method’s name and the number and types of arguments/parameters are called the method’s **signature**
- Overloading can be applied to any methods – void methods, methods that return a value, static methods, non-static methods, or to any combination of these

## Another Example

- The four set methods of the `Pet` class can be replaced with one overloaded set method with four different signatures, as follows:

```
public void set(String newName, int newAge,
                                        double newWeight) {
  name = newName;
  age = newAge;
  weight = newWeight;
}
```

## Another Example

```
public void set(String newName) {
   name = newName;//age, weight are unchanged
}
public void set(int newAge) {
   age = newAge; //name, weight are unchanged
}
public void set(double newWeight) {
   weight = newWeight;//name, age unchanged
}
```

## Another Example

- These methods then can be used in a client class as follows:

```
Pet myPet = new Pet();
myPet.set("Jack", 2, 5.5);
myPet.writeOutput();
myPet.set("Rex");    // Changing Name
myPet.writeOutput();
myPet.set(5);        // Changing age
myPet.writeOutput();
myPet.set(54.0);     // Changing Weight
myPet.writeOutput();
```

Murdoch UNIVERSITY

## Overloading and Automatic Type Conversion

- Note that Java can allow integers to be used where doubles are expected. Eg:

```
double d = 3.54 + 7;
```

- The statement on the previous page changes the weight of the pet to 54.0 pounds:

```
myPet.set(54.0);
```

- Suppose we forgot to include the decimal point and the zero, and wrote:

```
myPet.set(54);
```

## Overloading and Automatic Type Conversion

- What is the result?
- Instead of changing the pet's weight, we have changed the pet's age to 54 years!
- The same would happen if you create a Pet object using the constructor

```
Pet myPet = new Pet(54);
```

## Overloading and Automatic Type Conversion

- In general:
  - Java tries to find an exact type match between arguments and parameters first
  - If it cannot find an exact match then it tries to convert the type of an argument according to strict rules about what is allowed to be converted to what. Eg,
    - **int**s can be converted to **double**s
    - Note that **double**s cannot be automatically converted to **int**s
  - If no methods match then you would receive a compile time error before the program was allowed to run

## Privacy Leaks: Class Type Instance Variables

- Up until now we have mostly defined classes with instance variables of primitive types or Strings
- However, real-world programs are usually more complicated
  - Eg: a form object may have instance variables which are buttons
  - Eg: a school object may have instance variables which are lists of children

## Privacy Leaks: Class Type Instance Variables

- If you are the creator of a class which has instance variables of class type, then beware of supplying methods which return these values (i.e. objects)
- Such methods may return a reference to a supposedly hidden data value
  - The client may then be able to use the reference to change the value
  - This may corrupt your records so that your class does not behave correctly for that client

## Privacy Leaks: Class Type Instance Variables

- The problem arises because the variables of a class type contain the memory address (reference) of where an object is stored in memory
- See the example in the text (8th ed) Listing 6.18 :
  - An Insecure Class - a  simplified version of which is produced below

## Example: Cadet Class

```
/** File: CadetClass.java
 Example of a class that does NOT correctly
 hide its private instance variable.
*/
public class CadetClass {
   private Pet myPet; // a Pet instance variable
   public CadetClass() {  // constructor
      myPet = new Pet("Guard Dog",5,75.0);
   }
```

Murdoch UNIVERSITY

## Example: Cadet Class

```
  public void writeOutput() {
     System.out.println(“My pet’s details: ");
     myPet.writeOutput();
  }
  public Pet getPet() {
  // returns reference to the object !!!
     return myPet;
  }
} // end class CadetClass
```

## Example: Client Class

```
/** File: Hacker.java
 Toy program to demonstrate how a programmer can
 access and change private data in an object of the
 class CadetClass.
*/
public class Hacker {
  public static void main(String[] args) {
     CadetClass starFleetOfficer =
                              new CadetClass();
     System.out.println("starFleetOfficer contains: ");
     starFleetOfficer.writeOutput();
```

## Example: Client Class

```
        Pet badGuy;
        badGuy = starFleetOfficer.getPet();
        badGuy.setPet("Dominion Spy", 1200, 500);   // !!!
        System.out.println("Security breach!!!");
        System.out.println("starFleetOfficer contains: ");
        starFleetOfficer.writeOutput();
        System.out.println("Pet not so private!");
    }
} // end class Hacker
```

## Example: Output

```
/** Output
starFleetOfficer contains:
Here are my pet's details:
Name:   Faithful Guard Dog
Age:    5 years
Weight: 75.0 pounds
```

Murdoch UNIVERSITY

## Example: Output

```
Security breach!!!
starFleetOfficer now contains:
Here are my pet's details:
Name:    Dominion Spy
Age:     1200 years
Weight: 500.0 pounds

Pet not so private!
*/
```

## Privacy Leaks: Class Type Instance Variables

- **Two possible solutions to privacy leaks:**
  1. Stick to simple problems which only require methods to return primitive values or Strings (or nothing) (Strings are ok because they can not be changed)
  2. Read more advanced books and find out about copy constructor and **cloning**
     - That is, making a copy of an object, which resides separately in memory but starts off with the same data values

Murdoch UNIVERSITY

## Enumerations

- Java offers an **enumerated data type** which you can use to restrict the contents of a variable to certain values (that you want)
- An enumeration lists the values that a variable can have, and its definition takes the following form:

```
enum MovieRating{
EXCELLENT,AVERAGE,BAD }
```

  - a semicolon at the end of an enumeration definition is not necessary; if there is one, it will be ignored

## Enumerations

- An enumeration acts as a class type
  - The compiler creates a class `MovieRating` which can be used to declare variables as follows:

```
MovieRating rating;
```

- The enumerated values are names of public static objects whose type is `MovieRating`
- We can assign a value to an enumerated variable as:

```
rating = MovieRating.AVERAGE;
```

## Enumerations

- Then the variable can be used in a switch statement:

```
switch (rating) {
case EXCELLENT:
  System.out.println(“Must see movie”);
  break;
case AVERAGE:
  System.out.println(“Movie OK, not great”);
  break;
case BAD:
  System.out.println(“Skip it!”);
} // end case
```

# Enumerations

- The values of an enumeration behave like named constants
- Another example:

```
enum Suit{
CLUBS,DIAMONDS,HEARTS,SPADES }
```

```
Suit s = Suit.DIAMONDS;
```

- The class `Suit` has several methods available including `equals()`, `compareTo()`, `ordinal()`, `toString()` and `valueOf()`

## Enumerations

- Eg:

```
s.equals(Suit.HEARTS) …
s.compareTo(Suit.HEARTS) …
s.ordinal() // returns position or ordinal
   // value of DIAMONDS in the enumeration
s.toString() //returns string “DIAMONDS”
Suit.valueOf(“HEARTS”) // returns object
                       //  suit.HEARTS
```

## Packaging

- A package is a collection of related classes which:
  1. May contain classes private to the package (supporting encapsulation) and
  2. Can readily be imported together for use by other classes (supporting re-use)
- We will see how Java implementers can set up their own packages

## Class Libraries

- A number of related classes are placed in a **package**
- A number of related packages are grouped into a **Library**
- However, many useful packages already exist and are ready for use by implementers as libraries
    - These include the standard class libraries

## Class Libraries

- The classes in the package **java.lang** are automatically available to any program
  - Eg, **java.lang.System**, **java.lang.Integer** and **java.lang.Math**
- Classes and interfaces from pre-existing libraries such as in the Java API (also called Java Library, and is part of the Java software Development Kit – SDK) can be imported into a Java program

## Class Libraries

- The Java API is organised into a set of packages, where each package contains a collection of related classes and interfaces
- Packages are actually directory structures used to organise classes and interfaces

## Why Use Packages?

- Packages are useful to programmers as they provide a mechanism for software reuse
- As programmers, our goal should be to create reusable software components so we are not required to repeatedly redefine code in separate programs
- Another benefit of packages is that they provide a convention for unique class names

## Why Use Packages?

- With thousands of Java programmers around the world, there is a good chance that the names you choose for classes will conflict with the names that other programmers choose for their classes

## Importing Packages and Classes

- Apart from the **java.lang** package, which is automatically imported by Java, all other packages and classes must be imported into your program
- Other classes need to be *imported* either on their own:

```
import packagename.classname;
```

- Or, with the whole package:

```
import packagename.*;
```

## Importing Packages and Classes

- Eg:
- // import class Random

```
import java.util.Random;
```

- // import class Date

```
import java.util.Date;
```

- // import package java.util

```
import java.util.*;
```

Murdoch UNIVERSITY

# The Power of Java is its Library of Packages

- There are also other important libraries apart from **java.lang**
- For example, the Java Generic Library (JGL) contains classes for many basic algorithms and data structures
  - Swing contains a huge variety of GUI classes
- As you get more experienced in Java programming you will be able to take advantage of these amazing Java packages

Murdoch UNIVERSITY

# The Power of Java is its Library of Packages

- Let's have a look at some handy Java packages:

  `java.io.*`
  - Provides classes that are fundamental to the design of the Java programming language
  - Contains classes for inputting data into a program and outputting the results of a program
  - Eg: Provides input and output streams, file operations, etc.

  `java.awt.*`
  - Contains all of the classes for creating user interfaces and for painting graphics and images

## The Power of Java is its Library of Packages

`java.applet.*`

- Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context. Applets for Internet applications

`javax.swing.*`

- Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms.  Graphic User Interface components
- JButton, JOptionPane, JTextBox, JCheckBox, etc.

Murdoch UNIVERSITY

## The Power of Java is its Library of Packages

- Eg: Class `JCheckBox` (in package `javax.swing`)

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.AbstractButton
                          |
                          +--
javax.swing.JToggleButton
                                |
                                +--
```

**`javax.swing.JCheckBox`**

## The Power of Java is its Library of Packages

```
java.beans.*
```

- Re-useable software components
  - Allows programmers to develop re-useable components to create powerful applications and applets for the Internet
- Others include:

```
java.net.*
java.security.*  // ,etc.
```

- Have a look at the Java API documentation to see how much is available

Murdoch UNIVERSITY

## Making Your Own Packages (optional)

- In order to use a class in a package
  1. The class must be declared to be part of the package
  2. The package must be in the right directory
  3. The class must be imported by the client class

- These three steps are further explained as follows:

## Making Your Own Packages (optional)

1. The ***first line*** of each class in the package must be the keyword package followed by the name of the package

   Eg: to declare class `A` as part of package `X.Y` put

   ```
   package X.Y;
   ```

   as the first line in the source file for `A`

   (If there is no such declaration then the class belongs to a package called the default package and all such classes belong to that same package)

## Making Your Own Packages (optional)

2. Make sure that all the `.class` files in the package are put in a directory

   Eg:

   `C:\myclassdirectory\X\Y`

   and that the operating system’s environment variable

   `CLASSPATH` includes `c:\myclassdirectory`

   Eg:

   ```
   CLASSPATH=
   c:\jdk1.7\lib;c:\myclassdirectory
   ```

Murdoch UNIVERSITY

## Making Your Own Packages (optional)

3. Put:

   ```
   import X.Y.A;
   ```

   or

   ```
   import X.Y.*;
   ```

   in the client source file.

   Then you can refer to class `A`

   or, in the client, refer to the class as

   ```
   X.Y.A
   ```

Murdoch UNIVERSITY

## Creating a Re-useable Class

- Define a public class. If the class is not public, it can be used only by other classes in the same package
- Choose a package name and add a package statement to the source code file for the reusable class definition
- Compile the class so it is placed in the appropriate package directory structure and make the new class available to the compiler and interpreter

## Creating a Re-useable Class

- Import the reusable class into the program and use the class

## End of Topic 5

Murdoch UNIVERSITY