CENG 211 – Programming Fundamentals

Introduction to Object Oriented Programming

Classes and Objects

- A class is a template for objects that you will create later.
- The class definition specifies both the data the objects will store and their behavior.
- You can create many objects from the same class with new expression.
- These objects are called instances of the class.
- You store references to the objects that you create in variables of reference type.

Classes and Objects

```
public class Message {
    private String messageText;

    public Message(String text) { messageText = text; }

    public String toString() { return “Message = “ + messageText; }
}

public class MessageApp {
    public static void main(String[] args) {
        Message m1 = new Message(“Hello”);
        Message m2 = new Message(“Classes and Objects”);

        System.out.println(m1.toString());
        System.out.println(m2.toString());
    }
}
```

Fields and Methods

- The class definition can contain variable declarations that will contain data for each one of the objects or static class data.
  - These are called *attributes* or *fields* of the class.
- The class definition also may contain function definitions.
  - These are called *methods* of the class
- The non-static fields and methods of a class are only accessible from an object of that class.
- Each object carries its own copy of non-static data.

Classes and Objects

```
public class Message {
      private String messageText;

      public Message(String text) { messageText = text; }

      public String toString() { return “Message = “ + messageText; }
}

public class MessageApp {
      public static void main(String[] args) {
            Message m1 = new Message(“Hello”);
            Message m2 = new Message(“Classes and Objects”);

            System.out.println(m1.toString());
            System.out.println(m2.toString());
      }
}
```

Private field contains the data specific to each Message object

Constructors initialize the private data to default values or from supplied parameters

Methods of the class can access private data

static Fields and Methods

- Static fields hold data that is accessible through the class name. Instance objects do not get a copy of the static fields.
- Static methods do not need an instance of the class, instead they are called using the class name.
- Static fields store class wide valid data.
- Static methods have access to only static fields and methods of the class.

final Variables and Fields

- When the of a attribute or field is not going to change (it is a constant), we can place the final modifier with the variable type.

```
final int N = 5;
```

- If you accidentally try to change the value of a final variable, the compiler will catch the error.
- There are also certain optimizations and operations that can only be performed on final variable and fields.
- If you declare a non-static field as final, you need to initialize it either in the declaration or the constructor.
- If you declare a static field as final, you have to initialize it in the declaration.

Example

```
public class Book {
    public static final String UNKNOWN_AUTHOR_NAME = “Anonymous”;
    public static int defaultPrice = 20;

    public String title;
    public String [] authorNames;
    public int price;

    public Book(String title);
    public Book(String title, String[] authorNames) { ……… }
    public void increasePriceByPercent(int percentIncrease) { …… }

    public static Book bookFromUrl(URL url) { ……… }
}
```

Example

```
String [] duneAuthors = { “Herbert, Frank” }
Book [] duneSeries = new Book[N_DUNE_BOOKS];

duneSeries[0] = new Book(“Dune”, duneAuthors);

URL dune2Url = new URL(“http://www.amazon.com” +
    “/Dune-Messiah-The-Chronicles-Book/dp/0441172695”);
duneSeries[1] = Book.bookFromUrl(dune2Url);

for (Book book: duneSeries)
    if (book != null)
        book.increasePriceByPercent(20);
```

Constructors

- Constructors are special methods in the class definition that are run automatically every time a new object is created.
- Constructors have the same name as the class name and have no return type (even void is not allowed as a return type).
- You should initialize non-static fields to sensible initial values in the constructor.
- Constructors can receive arbitrary parameters much like any other method. You pass these parameters after the class name in the new expression.

this Reference

- Non-static methods of a class (including constructors) has access to a special reference variable named this.
- It points to the object that the method has been called on.
- You can use it whenever you need a reference to the object:

```
public class Book {
    public boolean isMember(Book [] series) {
        for (Book book: series)
            if (book == this) return true;
        return false;
    }
}
```

- You can access class fields and methods from the **<u>this</u>** variable. This is especially useful when a parameter name hides one of the fields:

```
public class Book {
    String title;
    public Book(String title) { this.title = title; }
}
```

Encapsulation

- The real reason for grouping data and methods in the same object is to create a scope that can restrict access to the class data:
  - Only the methods of the class should have direct access to the internal representation of the class data.
  - Code creating and using objects of the class should access/ manipulate object data only by calling appropriate methods on the object.
- This is the principle of encapsulation.

Public/Private

- If a class/field/method is declared **public**, it is accessible from any code.
- If a class/field/method is declared **private**, it is accessible only within the top-level class definition.

- If it is not declared public or private, it has **default** access (it is accessible only within the same package, which is covered later in these notes).
- We will later talk about **protected** access when we discuss inheritance.

Public/Private

- Generally,
  - We will declare all attributes as private. This will restrict access to these fields only within the class definition.
  - We will write public methods that allow users of the class to manipulate and query objects.
  - We will write some private utility methods that are only called from public and private methods in the same class.

Example:

```
// How are the coordinates stored? Cartesian/Polar/…

public class Point {
    private float [] coordinates;

    publicPoint(float x, float y) {
        coordinates = new float[2];
        setCoordinates(x, y);
    }

    public void setCoordinates(float x, float y) { … }
    public String toString() { … }
    public float distanceTo(Point p) { … }
    public void rotate(float angle);
}
```

Encapsulation and Design Invariants

- One of the goals of encapsulation is to protect the relationships among attribute values from erroneous manipulation.
- Public fields can be changed by any piece of code at any time and independent of each other. This can lead to all kinds of unexpected results and bugs.
- If these relationships hold true all the time, we can predict code behavior more easily and we will have less bugs.
- Such relations that must hold true (constraints) between attributes are called invariants. They are an important part of class design and a useful tool for software verification.

Example:

```
public class UnitVector2D {
    private float x;
    private float y;

    public UnitVector2D() { x = 1.0f; y = 0.0f; }

    public void setCoordinates(float x, float y) {
        float normRequested = norm(x, y);
        if (normRequested > 0) {
            this.x = x / normRequested;
            this.y = y / normRequested;
        } else {
            ……… // throw an error
        }
    }

    private float norm(float x, float y) { … }
}
```

Get/Set Methods

- Sometimes, you want to expose the object data in a specific format.
- In this case, instead of creating public fields, add a couple of methods that
  - start with "get" that return the data in the specific format. These are called getters.
  - start with "set" that set the data from parameters in the specific format. These are called setters.
- Since getters and setters can do arbitrary checks and computations, they can ensure that invariants of your design are protected.

Example:

```
public class Point {
    private float [] coordinates;

    publicPoint(float x, float y) {
        coordinates = new float[2];
        setCoordinates(x, y);
    }


    ………
    public float getX() { … }
    public float getY() { … }
    public void setX(float x) { … }
    public void setY(floay y) { … }
}
```

Data Classes

- Sometimes, you will want to create a data structure that will hold several fields in a fixed representation.
- There is nothing fundamentally wrong with this, you can create and use C like structures in Java:

```
class PointF {
    public float x;
    public float y;
}
```

- If you create data classes, limit their methods to simple constructors that initialize the fields.
- When and how much to encapsulate is a design decision, you will get better at it as you work on larger projects.

Method Overloading

- In Java, you can have multiple versions of a method with different parameter lists (either in type/number/or both).
- Depending on the method call signature, the more suitable one will be called:

```
public class Vector {
    .........
    public void multiply(int scale) { … } // x *= scale
    public void multiply(float scale) { … } // x *= scale
    public void multiply(Vector v) { … } // x *= v
    public void multiply(float scale, Vector v) { … } // x *= scale * v
}
```

- When listing overloaded methods, list them from the most general to the most specific.

Method Overloading

- Do not over-use overloading! Think about how the code will look like at call time.
- When implementing overloaded methods, try to implement the most general one and have the others call it.

```
public class Vector {
    .........
    public void multiply(Vector v) {
        multiply(1.0f, v);
    }

    public void multiply(float scale, Vector v) { … }
}
```

Overloading Constructors

- One of the most common methods to overload are the constructors:
- This way you can create objects in different ways.

```
public class Vector {
    private float [] data;

    public Vector(int n);
    public Vector(float [] v);
}
```

Overloading Constructors

- When implementing overloaded constructors, you can still call the other constructors you have already implemented by calling this as a function:

```
public class Vector {
    private float [] data;

    public Vector(int n) {
        data = new float[n];
    }

    public Vector(float… v) {
        this(v.length);
        for (int i = 0; i < v.length; ++i)
            data[i] = v[i];
    }
}
```

Garbage Collection

- When there are no references to an object, Java Run-time automatically frees up the object. This is called garbage collection.

- You do not need to explicitly free objects that you create.

- Freeing of objects do not happen instantaneously, the garbage collector runs at certain times (that you can not easily guess) and frees up all objects without references.

Packages

- Java classes are organized into packages.
- Even when you do not declare a package name, your class definitions are placed in a *nameless* package.
- A package component (for example a class) have direct access to all other package components and public components of the java.lang package.
- Organizing code in packages prevents name conflicts.
- To access classes in other packages you need to either import them or prefix the class name with the package name.

Packages

- To place the contents of a file in a certain package, you use the package keyword followed by the package name. This should be the first thing in a file.

- It is customary to create package names that are reversed domain names of your institution/company to avoid clashes in the package names.

- Files in a package are placed in a directory hierarchy that has a subfolder for each part of the package name. A class Vector in package tr.edu.iyte is placed on a path tr/edu/iyte/Vector.java

Packages

- Package Encapsulation
  - If you do not specify any access modifier like public/private, a class/attribute has package access, it is visible anywhere from the same package.