Objects and Classes

Introduction to Object Oriented Programming (OOP)

- Class in Java contains variables and methods.
- The real purpose of a class in Java is to implement/model an object that contribute to the solution of the problem.
- Programming methodology before ~1980: use the modular programming technique to help build **large-scale** complex computer programs.
- Today's methodology: use the **object** concept to build large-scale complex computer programs. This style of programming using object is called the **Object-Oriented Programming (OOP)**
- How OOP help us write complex programs:
  - **Abstraction**: OOP provides abstract classes to help reduce(=hide) details
  - **Inheritance**: allows existing code to be re-used.
  - **Polymorphism**: allows existing code to be modified/enhanced.
  - **Encapsulation**: prevents code in other classes from accessing/modifying important variables to localize debugging.

> **Object**: an object represents an entity in the real world that can be distinctly identified.

- An object has:
  - A unique identity
  - A state
  - A behavior

> The **state** of an object (also known as its properties or attributes) is represented by data fields with their current values.

- A Java class represents the state/properties of objects using:
  - The **instance variables** inside a class
  - Each object will have its own instance variables.

> The **behavior** of an object (also known as its actions) is defined by methods.
> To **invoke** a method on an object is to tell the object to perform an action.

- A Java class defines the behavior of objects using:
  - The **instance methods** inside a class

    - All objects of a class share the instance methods (because they have the same behavior).
- A class is used as a template(=description) to construct the object's data fields and to define its methods:
    - When we create objects of a class, Java will use the class definition to allocate the instance variables for that object.
    - When you invoke some method on an object, Java will run the code in the method definition on the instance variables of the object.
    - We can create as many instances(=objects) of a class as we need:
        - Each object will have its own properties(=instance variables).
        - But all objects will share the same actions(=instance methods).

Defining a Class & Creating Objects

```
public class Circle {
    public double radius = 1; // The radius of this circle

    public Circle() { } // constructor 1 for a circle object
    public Circle(double newRadius) { // constructor 2 for a circle object
        radius = newRadius;
    }

    public double getArea() { // return the area of this circle
        return 3.14159 * radius * radius;
    }

    public void setRadius(double newRadius) { // set new radius for this circle
        radius = newRadius;
    }
}
```

We use the `Circle` class to create two `Circle` objects:

```
    public static void main() {
        Circle circle1 = new Circle(); // Invokes Circle() to make this circle

        Circle circle2 = new Circle(2); // Invokes Circle(double) to make this circle

        double area1 = circle1.getArea(); // Tell circle1 to run getArea()
        System.out.println("Area1: " + area1);

        double area2 = circle2.getArea(); // Tell circle2 to run getArea()
        System.out.println("Area2: " + area2);

        circle1.setRaius(5); // Tell circle1 to run setRadius()

        double area1 = circle1.getArea(); // Tell circle1 to run getArea()
        System.out.println("Area1: " + area1);
    }
```

- See `TestCircle.java` and `Cirlce.java`

> **Unified Modeling Language (UML)**: a standardized modeling representation description of classes and objects.

- A Java class uses **variables** to define data fields (properties) of objects.
- A Java class uses **methods** to define the actions/behaviors of objects.
  - Methods to define the actions of objects DO NOT have the `static` qualifier
- A class provides special method called **constructors** which are invoked only to create a new object.
  - Constructors are designed to perform initializing actions, such as initializing the data fields of objects.
- A class that represents real world objects usually does not need a `main()` method. Without a `main()` method, such class cannot be run as a Java program.
  - Though we may include a `main()` method in the class to test the methods, but it is preferred to write a separate class to do the testing.
- Preventing undesirable behavior in objects:
  - The `Circle` class implementation allows a user to access the object variables directly because we did not define `radius` to be private.

```
public class Circle {
  public double radius = 1; // Then radius cannot be modified outside the class

  public static void main() {
      Circle circle1 = new Circle();
      circle1.radius = 10; // changes the value of radius directlly
  }
}
```

- We prevent direct access to variables in a class by using the `private` qualifier.

```
public class Circle {
  private double radius = 1; // Then radius cannot be modified outside the class

  public static void main() {
      Circle circle1 = new Circle();
      circle1.radius = 10; // complie error
  }
}
```

Constructors of a Class

> **Constructors** are special methods in a class that is only invoked when an object is created using the `new` operator:
>
> ```
> ClassName objVar = new ClassName(...);
> ```

- Constructors have 3 special properties:
  - A constructor must have the same name as the class itself.
  - Constructors do not have a return type - not even `void` .
    - If we include a `void` return type, then the method is not a constructor, but a behavior that the object can take.
  - Constructors cannot be invoked like an ordinary method.
- Like regular methods, constructors can be overloads (i.e., multiple constructors can be defined with different signatures).
- Rules on constructors and the default constructor:
  - Every class must have at least one constructor.
  - If a class does not have any constructor, then the Java compiler will automatically insert this constructor: `className() { }` . This constructor is called the **default constructor**.

- However, the Java compiler will not insert the default constructor if there is a constructor defined in the class.

Objects as Reference Data Types

```
Circle  is a reference data type
circle1 is a reference variable
circle1 references (points to) a Circle object
```

- We create variables to store the properties of a new object when we create the object
- The behavior of an object (=program instructions) is stored when Java compiles the class definition.
- An object's member can refer to:
  - A data field in the object
  - A method in the object
- After an object is created, its data can be accessed, and its methods can be invoked using the `dot` operator.

  ```
  objectRefVar.dataField  references a data field in the object
  objectRefVar.method(arguments)  invokes a method on the object
  ```

- The `dot` operator is also known as the object member access operator.
- Why Java have reference typed variables and primitive typed variables?
  - Variables of a primitive data type can only store 1 value but can be accessed quickly -- such variables are mainly used in computations.
  - Objects can have many data fields and can allow the programmer to represent complex things in the real world.
    - Objects are mainly used for data representation
    - Accessing to data in an object is slower (need 2 memory accesses)
- We can access the member variable without using any reference variable:
  - An instance method is always invoked using an object reference variable:
    `objectRefVar.method(arguments)`
  - The variable `objectRefVar` is also passed to an instance method as an implicit (=hidden) parameter. The name of the implicit parameter is called `this` .
-
  - See `Circle.java`
- This implicit parameter `this` is almost never necessary to write in a Java class. There is only 1 case that it is necessary:
  - when a parameter variable has the same name as an instance variable in the class.

- ◦ See `Circle.java`
- The `this` keyword is can also be used to invoke another constructor of the same class.

Copying Objects

- **Copy** an object means:
  - Make a duplicate of an object where the duplicated object contains the same data as the original object.
  - Updating the instance variables in the duplicate object must not affect the values in the original object.
- One way is to create a new object and then copy the data fields.

```
public static void main() {
  Circle circle1 = new Circle(4);

  // Make a COPY of circle1
  Circle circle2 = new Circle();
  circle2.radius = circle1.radius;
}
```

- ◦ See `CircleCopy.java` . This method only works when the data fields are defined in `public` .
- Another way is through a copy constructor:

```
public class Circle{
  private double radius = 1;
  public Circle() { } // constructor for a circle object

  public Circle(Circle c) { // copy constructor that copies circle c
    radius = c.radius;
  }
}
```

To invoke the copy constructor:

```
public static void main() {
  Circle circle1 = new Circle(4);
  Circile circle2 = new Circle(circle1);
}
```

- ◦ See `CircleCopy.java` .

Arrays of Objects

- Similar to doubles and integers, we also have arrays of objects in Java. They are also defined in a similar way.
- In other words, we can create a `Circle` object with `new` and assign it to an array element

```
Circle[] circleArray = new Circle[10];
circleArray[0] = new Circle(4);
```

- However, an array of primitive variables is different from an array of reference variables.
  - Primitive:
    - After creating an array of primitive variables, each array element can store a value.
    - Primitive type array variables (`number[k]`) contains values and is used in computations
  - Reference:
    - After creating an array of reference variables, each array element can store a reference of an object.
    - Reference array variables (`circleArray[k]`) contains references and is used with the member selection operator `.` (the `dot` operator).

Data Field Encapsulation

- The most important application of visibility(=accessibility) modifiers is: **data field encapsulation**.

> **Data Field Encapsulation** is making data fields in an object inaccessible (= `private`) to other classes (which will disallow other classes from using the data fields directly).

- Encapsulation is important because
  - If a data field is not `private` , program written by other programmers can tamper with the data fields.
  - When other programs use a data field in an object directly, changing the implementation of the object is more difficult.
    - Changing the implementation of the object means change the way we present the properties of an object.
      - For example, we can use `String` to represent `suit` as `{"Spades", "Hearts", "Diamonds", "Clubs"}` . Meanwhile, we can also use `int` to represent it as `{0 = "Spades", 1 = "Hearts", 2 = "Diamonds", 3 = "Clubs}` .
    - When we use `String` , we can use `card.suit.compareTo("Spades") == 0` to test if the suit of the card is spade. However, if we change the implementation of card to

`int`, the same code `card.suit.compareTo("Spades") == 0` will cause an error because we do not have a `.compareTo()` method for an integer.

- So, data field encapsulation requires that data fields are defined as `private`.
  - When other classes need to read a data field, we must provide a `public` mutator method.
    - See `CardPrivate.java` and `TestCardPrivate.java`.
  - When we change the implementation of an object, we can still maintain compatibility with existing Java program by **providing updated accessor/mutator methods** that achieve the same effect as the old implementation.

> **Immutable Objects**: An immutable object is an object where its properties cannot be changed after it is created.

- Why we what to have immutable objects:
  - Some computer applications are used to record a history of events which are represented by objects
  - The "historical objects" must not be changed.
- To prevent the data fields of the objects being updated:
  - Prevent the variables being updated with direct access (e.g. `circle1.radius = newRadius`):
    - Define all distance variables as `private`.
  - Prevent the variables being updated with a mutator method:
    - Immutable objects must not have any mutator methods.
  - Prevent the variables being updated with a reference variable:
    - Immutable objects should not have accessor methods that return a reference to an object that has `public` data fields.
- An example of immutable object: the `String` class in Java will create immutable `String` objects:
  - The `String` class only has methods that construct a new `String` from an input string, and the input string is not updated.
  - Example:

```
public static void main(String[] args) {
  String s1 = "abc";
  String s2 = s1.toUpperCase();

  System.out.println(s1); // "abc", unchanged
  System.out.println(s2); // "ABC"
}
```

Passing Objects as Parameters to Methods

- Methods can have reference type parameter variables.
- - See `TestCircle.java`
- However, the following code will change the properties of the object directly:

```
public static void incrementRadius(Circle c) {
  c.radius++; // Increment radius by 1
}

public static void main(String[] args) {
  Circle circle1 = new Circle(4);
  System.out.println(circle1.getRadius()); // 4
  incrementRadius(circle1); // radius of circle1 increases by 1
  System.out.println(circle1.getRadius()); // 5
}
```

  - In Java, the formal parameter `c` is an alias of the actual parameter `circle1`. So, `c.circle++` will also update `circle1.radius`.
  - - See `CopyReference.java`
- Review: Passing primitive variables to methods
  - In Java, the value of the argument copied (=assigned) to the parameter variable. So, `x` in `main()` and `c` in `increment()` are different variables.
  - When `increment()` executes `c++`, it updates the parameter variable `c`.
  - The variable `x` in `main()` is not affected.
- Passing reference variables to methods
  - The reference type `Cricle` variable `x` contains a reference to a `Circle` object.
  - In Java, the value of the argument copied(=assigned) to the parameter variable. `x` in `main()` and `c` in `increment()` both reference to the same `Circle` object.
  - When `increment()` executes `c.radius++`, it updates the `radius` variable through the reference `c`.
  - The variable `x.radius` in `main()` is ALSO affected because it is the same object.

```
public static void main(String[] arg) {
  Circle circle1 = new Circle(4);
  System.out.println(circle1.getRadius()); // 4.0
  updateCircle(circle1);
  System.out.println(circle1.getRadius()); // 4.0
}
public static void updateCircle(Circle c) {
  c = new Circle(99);
}
```

- The reference type `Circle` variable `circle1` contains a reference to a `Circle` object.
- `circle1` in `main()` and `c` in `update()` both refer to the same `Circle` object.
- When `update()` executes `c = new Circle(99)`, it creates another `Circle` object and assign its address to reference variable `c`.
- The variable `circle1.radius` in `main()` is not affected.
- Through this example, we know: we can never make `x` in `main()` refer to a different object using a method call. This is because `x` is passed-by-value, we cannot update `x` and make it refer to a different object.
- If we really want to write a method to update the reference of `x`, here's an example to do so:

```
public static void main(String[] arg) {
  Circle circle1 = new Circle(4);
  System.out.println(circle1.getRadius()); // 4.0
  circle1 = updateCircle(circle1); // Step 2
  System.out.println(circle1.getRadius()); // 99.0
}
public static Circle updateCircle(Circle c) {
  c = new Circle(99);
  return c; // Step 1
}
```

Static Variables (and Constants) and Static Methods

- There are 2 kinds of variables that can be defined inside a class (that is outside any method):

```
public class Circle{
  public double radius; // (1) an instance variable
  public static int count; // (2) a "static" variable
}
```

- Instance variables and `static` variables of objects are different:
  - Each object has its own copy of an instance variable.
  - `static` variable belongs to the class and all objects of that class share the same copy of a `static` variable.
  - In other words, there is only 1 only of a `static` variable in a Java program.

```
public static void main(String[] args) {
  CircleCount circle1 = new CircleCount(2);
      CircleCount circle2 = new CircleCount(4);

      circle1.count = 99;

      System.out.println(circle1.radius); // 2.0
      System.out.println(circle1.count); // 99
      System.out.println(circle2.radius); // 4.0
      System.out.println(circle2.count); // 99

      circle1.radius++; // Updates an instance variable
      circle1.count++; // Updates a static variable
      System.out.println(circle1.radius); // 3.0
      System.out.println(circle1.count); // 100
      System.out.println(circle2.radius); // 4.0
      System.out.println(circle2.count); // 100
}
```

  - `circle1.count` and `circle2.count` are always the same because static variables are shared
  - `circle1.radius` and `circle2.radius` are independent to each other because instance variables are not shared.
- Applications of `static` variables:
  - The most common application where we need to use a `static` variable in a class is when writing a class that can keep a count on the number of objects that has been created by a Java program.
  - How to implement?
    - Define a `static` variable named `count` and initialize it to zero.
    - Each constructor of the class must increase the `count` variable by one.
  - Why it works?
    - Because when an object is created, some constructor method is invoked once, and this algorithm will keep track on the number of objects created.
  - Example:

```
public class Circle{
  public double radius = 1;
  public int count = 0;

  public Circle() {
    count++;
  }
  public Circle(double newRadius) {
    radius = newRadius;
    count++;
  }
}
```

- There are also two kinds of methods that can be defined inside a class: instance method and `static` method.
  - Instance methods always have an implicit(=hidden) object reference parameter (`this`) and can access instance variables.
  - `static` method do not have an implicit(=hidden) object reference parameter and cannot access instance variables.
- Properties of `static` methods:
  - A `static` method belongs to a class. For this reason, `static` methods are also known as class methods.
  - A `static` method can be invoked without using an object instance: `Math.pow(x, n)`
  - `static` methods can only access `static` members:
    - Invoke other `static` methods
    - Access `static` variables
    - `static` methods cannot access any instance variables nor invoke instance methods.
- `static` methods are used to perform a task that is not associated with a particular object.
- Instance methods are used to perform a task using data in a specific object.
- `static` methods can be invoked in 2 different ways:
  - `instanceVar.staticMethod(...)`
  - `ClassName.staticMethod(...)` `<-- Preferred`
- Some classes may have useful constants defined in them (such as π and e). Since a constant cannot change its value, we will only need one copy of it, and so a constant can always be defined as `static` .
- The `static` block
  - A `static` block is a nameless and parameterless `static` method in a class:

```
public class myClass {
  ...  (other memebers omitted for brevity)

  // A static block
  static
  {
    ... (statements)
  }
}
```

- Use of a `static` block:
  - `static` blocks are executed before the `main()` method
  - `static` blocks are used to initialize `static` variables in a class.