Chapter 6: Introducing Classes

Welcome to the world of **Object-Oriented Programming (OOP)**! In this chapter, we will explore the backbone of OOP — **Classes** and **Objects**. Think of a class as a blueprint and objects as the real-world instances created from that blueprint.

---

1. The General Form of a Class

A **class** is a blueprint for creating objects. It combines **data** (variables) and **methods** (functions) to define an entity's state and behavior. Here's the general syntax:

```
class ClassName {
    // Instance variables
    type var1;
    type var2;

    // Methods
    returnType methodName(parameters) {
        // Method body
    }
}
```

Key Points:

- **Instance variables**: Represent the data (state) of an object.
- **Methods**: Define the behavior (actions) of an object.

---

2. A Simple Class

Let's look at a basic example:

```
class Box {
    double width;
    double height;
    double depth;
}
```

This class defines a **Box** with three dimensions but no behaviors yet. It's like defining a data container.

---

3. Declaring Objects

Objects are created using the `new` **keyword**. Here's how we create an object from a class:

```
Box myBox = new Box();
```

- `Box` is the class name.
- `myBox` is the reference variable (object).
- `new Box()` creates a new object in memory.

---

4. A Closer Look at `new`

The `new` keyword does two things:

1. Allocates memory for a new object.
2. Calls the **constructor** to initialize the object.

Without `new`, you'd only have a reference, not an actual object.

---

5. Assigning Object Reference Variables

Object reference variables store the address of the object in memory. When you assign one object reference to another, both variables point to the same object.

```
Box box1 = new Box();
Box box2 = box1;  // Now, box2 and box1 refer to the same object
```

Changes made through `box2` will also reflect in `box1`.

---

6. Introducing Methods

Methods define the behavior of a class. They perform operations, often on the data stored in instance variables.

Example:

```
class Box {
    double width, height, depth;

    void displayVolume() {
        System.out.println("Volume: " + (width * height * depth));
    }
}
```

Here, `displayVolume()` is a method that calculates and displays the volume of the box.

---

7. Adding a Method to the Box Class

Let's enhance our `Box` class by adding functionality:

```
void volume() {
    System.out.println("Volume: " + (width * height * depth));
}
```

Now, every `Box` object can call this method to calculate its volume.

---

8. Returning a Value

Methods can return values to the caller. For example:

```
double volume() {
    return width * height * depth;
}
```

This method returns the calculated volume, which can be stored or used elsewhere:

```
double vol = myBox.volume();
```

---

9. Adding a Method That Takes Parameters

We can create methods that accept parameters to perform operations with different values:

```
void setDimensions(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
}
```

This allows us to set dimensions dynamically for any `Box` object.

---

10. Constructors

A **constructor** is a special method called when an object is created. It has the same name as the class and **no return type**.

```
class Box {
    double width, height, depth;

    Box() {  // Default constructor
        width = height = depth = 0;
    }
}
```

When you create an object with `new Box();`, the default constructor initializes the object.

---

11. Parameterized Constructor

A constructor can accept parameters to initialize objects with specific values:

```
Box(double w, double h, double d) {
    width = w;
    height = h;
```

```
    depth = d;
}
```

Now, when you create a `Box` object, you can pass values:

```
Box myBox = new Box(10, 20, 30);
```

---

12. The `this` Keyword

The `this` keyword refers to the current instance of the class. It is used to resolve conflicts between instance variables and parameters with the same name:

```
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Here, `this.width` refers to the instance variable, while `width` refers to the parameter.

---

13. Instance Variable Hiding

Instance variables can be hidden by local variables or method parameters with the same name. The `this` keyword helps access the hidden instance variables.

---

14. Garbage Collection

Java provides **automatic memory management** through **Garbage Collection**. Objects that are no longer referenced are automatically destroyed, freeing up memory.

You can request garbage collection manually using:

```
System.gc();
```

However, this is not guaranteed to run immediately.

---

15. A Stack Class

To understand the practical application of classes and methods, let's implement a **Stack** data structure:

```
class Stack {
    int[] stack = new int[10];
    int top;

    Stack() {
        top = -1;
    }
```

```
    void push(int item) {
        if (top == 9) {
            System.out.println("Stack is full");
        } else {
            stack[++top] = item;
        }
    }

    int pop() {
        if (top < 0) {
            System.out.println("Stack is empty");
            return 0;
        } else {
            return stack[top--];
        }
    }
}
```

Explanation:

- `push()` adds an item to the stack.
- `pop()` removes the last item.

Summary

- **Classes** encapsulate data and methods, forming the foundation of OOP.
- **Objects** are instances of classes.
- **Constructors** initialize objects automatically.
- **Garbage Collection** helps manage memory efficiently.
- Methods can return values, accept parameters, and define behavior.
- The `this` keyword resolves conflicts between local and instance variables.

By mastering these concepts, you'll have a solid foundation in OOP principles.