Polymorphism (and other items)

Reading

- All of Chapter 11, with the exception of
  - Dynamic binding – Section 11.8
  - Custom Stack class – Section 11.12
Objectives

- To distinguish differences between overriding and overloading (§11.5).
- To understand advantages of polymorphism
- To describe casting and explain why explicit downcasting is necessary (§11.9).
- To explore the equals() method in the Object class (§11.10).
- To store, retrieve, and manipulate objects in an ArrayList (§11.11).
- To prevent class extending and method overriding using the final modifier (§11.14).

Overriding vs. Overloading

```java
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overrides the method in B
    public void p(double i) {
        System.out.println(i);
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overloads the method in B
    public void p(int i) {
        System.out.println(i);
    }
}
```
Polymorphism

Method m takes a parameter of the Object type. You can invoke it with any object.

Polymorphism - An object of a subtype can be used wherever its supertype value is required.

When the method m(Object x) is executed, the argument x's toString method is invoked.

x may be an instance of
- GraduateStudent,
- Student,
- Person, or Object.

Casting Objects

- Previously used for primitives
- For objects => convert an object of one class type to another within an inheritance hierarchy, for example

```java
public static void m(Object x) {
    System.out.println(x.toString());
}
...
m(new Student());
```

assigns the object new Student() to a parameter of the Object type

- This statement is equivalent to:

```java
Object o = new Student(); // Implicit casting
m(o);
```

The statement is legal because an instance of Student is automatically an instance of Object.
Explicit Casting

- If o is of type Object
  \[ \text{Student } b = o; \]  
  \text{Compilation Error}

- Correct - \text{Object } o = \text{new Student()}
- Incorrect - \text{Student } b = o
  - Student object is always an instance of Object
  - Object is not necessarily an instance of Student.
- Solution - use an explicit casting.
  \[ \text{Student } b = (\text{Student}) o; \] // Explicit casting

Casting from Superclass to Subclass

- Explicit casting must be used when casting an object from a superclass to a subclass.
- This type of casting may not always succeed.

  \[ \text{Apple } x = (\text{Apple}) \text{fruit}; \]

  \[ \text{Orange } x = (\text{Orange}) \text{fruit}; \]
Example

- An apple is a fruit
  - you can always safely assign an instance of Apple to a variable for Fruit
- However, a fruit is not necessarily an apple
  - you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

The `instanceof` Operator

- Use the `instanceof` operator to test whether an object is an instance of a class:

```java
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
    System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
    ...
}
```
The `equals` Method

- Compares the contents of two objects.
- Default implementation
  ```java
  public boolean equals(Object obj) {
      return (this == obj);
  }
  ```
  Only checks if the identifiers point to the same object

Overridden in the Circle class.

```java
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    } else {
        return false;
    }
}
```

Warning

- Use the comparison operator (==) to compare primitives
- Use the equals method to compare objects
The ArrayList and Vector Classes

You can create an array to store objects. But the array’s size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.

```
java.util.ArrayList
+ArrayList(): void
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): Object
+set(index: int, o: Object): Object
```

- Creates an empty list.
- Appends a new element \( o \) at the end of this list.
- Adds a new element \( o \) at the specified index in this list.
- Removes all the elements from this list.
- Returns true if this list contains the element \( o \).
- Returns the element from this list at the specified index.
- Returns the index of the first matching element in this list.
- Returns true if this list contains no elements.
- Returns the index of the last matching element in this list.
- Removes the element \( o \) from this list.
- Returns the number of elements in this list.
- Removes the element at the specified index.
- Sets the element at the specified index.

Polymorphism in Action

- Section 11.11 of the text introduces the ArrayList, and in Listing 11.8, we have the code
- ```java
  java.util.ArrayList cityList = new java.util.ArrayList();
  ```
- Better to code
  ```java
  import java.util.*;
  ArrayList cityList = new ArrayList();
  ```
- Listing 11.8 uses the following methods on cityList: add, contains, indexOf, isEmpty, remove, and get
- These are methods of List, therefore much better code is
  ```java
  List cityList = new ArrayList();
  ```
List Vs. ArrayList

List
- add
- get
- set
- isEmpty
- remove
- size
- etc.

ArrayList
- Overridden methods (e.g., get)
- Inherited methods (e.g., removeAll)
- Additional methods (e.g., trimToSize)

Generic Programming

- Significant change to Java (with Java 5.0)
- Promotes reusability and code safety – reusable code can be written for objects of many different types
- Example

```java
public class ArrayList {  //before Java 5.0
    public Object get(int i) { ... }
    public void add(Object o) {...}
    ...
    private Object[] elementData;
}
```

No compile time error checking (e.g., add method) and a cast is necessary when you retrieve a value

1. More than text
Generic Type Parameters

- You can now specify a type parameter (not required)
  
  ```java
  ArrayList<String> files = new ArrayList<String>();
  ```

- Advantages
  - Code is easier to read
  - No cast is required for calling get
  - Compiler can provide additional compile-time error checking

The protected Modifier

- The protected modifier can be applied on data and methods in a class.

- A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
Note

• Modifiers are used on classes and class members (data and methods)
• the “final” modifier can also be used on local variables in a method.
• A “final” local variable is a constant inside a method.

The final Modifier

• The final class cannot be extended:
  ```java
  final class Math {
      ...
  }
  ```

• The final variable is a constant:
  ```java
  final static double PI = 3.14159;
  ```

• The final method cannot be overridden by its subclasses.
Have You Satisfied the Objectives

- To distinguish differences between overriding and overloading (§11.5).
- To understand advantages of polymorphism.
- To describe casting and explain why explicit downcasting is necessary (§11.9).
- To explore the `equals()` method in the `Object` class (§11.10).
- To store, retrieve, and manipulate objects in an `ArrayList` (§11.11).
- To prevent class extending and method overriding using the `final` modifier (§11.14).