Topic 4

Inheritance
Objectives

- To learn about the concept of *inheritance*
- To understand how to *inherit* and *override* methods from a *superclass*
- To learn about *inheritance hierarchies* and the general superclass *Object*
- To learn about *casting* objects
- To learn about the *instanceOf* operator
Inheritance

• **Inheritance**: a mechanism for deriving a new class from an existing one

• **Motivation**:
  • Can *reuse* existing classes
    • Faster and cheaper than writing them from scratch
  • Can *organize* classes in a hierarchical manner
    • e.g. can go from more general to more specific classes
Example of a Class Hierarchy

Vehicle

- Car
  - SUV
  - Smartcar
  - Van
- Bus
  - Schoolbus
  - LTCbus
  - Greyhound
Example of a Class Hierarchy

Shape

2DShape
- Circle
- Rectangle
- Triangle
- Square

3DShape
- Sphere
- Cube
- Tetrahedron
Example of Inheritance

• Suppose we already have a class called BankAccount
  • There are *specialized* types of bank accounts, such as savings accounts and checking accounts
  • So, we can write new classes called SavingsAccount and CheckingAccount that are *derived from* the BankAccount class (the *base* class)
# More Examples of Inheritance

<table>
<thead>
<tr>
<th>Base class</th>
<th>Derived Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangle</td>
<td>Square</td>
</tr>
<tr>
<td>Student</td>
<td>UndergradStudent</td>
</tr>
<tr>
<td></td>
<td>GradStudent</td>
</tr>
<tr>
<td>Loan</td>
<td>CarLoan</td>
</tr>
<tr>
<td></td>
<td>StudentLoan</td>
</tr>
<tr>
<td></td>
<td>MortgageLoan</td>
</tr>
</tbody>
</table>
Inheritance Terminology

• The derived new class is called the **subclass**
  • Also called the *child* class or *derived* class
• It inherits the attributes and methods of the **superclass**
  • Also called the *parent* class or *base* class
• It can add new attributes or methods for itself, *i.e.* it can **extend** the parent class
  • In fact, the Java keyword to make a subclass is **extends**
Java Example of Inheritance

/* Rectangle.java: a class that models a rectangle */

public class Rectangle {
    private int length;
    private int width;
    public Rectangle(int rLength, int rWidth) {
        length = rLength;
        width = rWidth;
    }
    public int getLength( ) {
        return length;
    }
}

// cont’d..
public int getWidth() {
    return width;
}
public int area() {
    return length * width;
}
public String toString() {
    return "Rectangle: " +
            "Length(" + length + ") " +
            "Width(" + width + ")";
}

    // end of class Rectangle
/ * Square.java: a class that models a square */

public class Square extends Rectangle {

    // no new attributes need be introduced

    public Square(int s) {
        // calls the 2-variable superclass constructor
        super(s, s);
    }

    public int getSide() {
        return getWidth();
    }

    public String toString() {
        return "Square: Side(" + getSide() + ")";
    }
}

Inheriting Visibility

• **public** variables and methods: children can access them directly *(except the constructor)*
• **private** variables and methods: children **cannot** access them directly
  • Why not? this would violate information hiding
• **protected** = may be accessed directly by any class in the same package, or by any subclass
  • So, children classes *can* access protected variables and methods of parent class directly
The **super Reference**

- **super** is a reserved word used in a derived class to refer to its parent class
- Allows us to access those members of the parent class that are *not* inherited
  - *Invoking the parent’s constructor*: the first line of a child’s constructor should be `super(...)`;
Is-a Relationship

• The derived class *is a* more specific version of the original class
• So, subclass object is of type *subclass*, but also it is an instance of *superclass*
  • *Example*: A Square object *is a* Rectangle
Discussion

• Why extend an existing class, *i.e.* why not just change the existing class by adding the new attributes and methods?

• Can you think of more examples of classes we can model with an inheritance relationship?
Example: BankAccount class

- Suppose we have a class `BankAccount` with attributes
  
  ```java
  private String accountNumber;
  private double balance;
  ```

  and public methods `deposit, withdraw, printBalance, getBalance, toString`

- What attributes and methods of the `BankAccount` class can be accessed 
  `directly` by code in its subclasses?
Example: BankAccount class

• What new attributes might we have in subclasses SavingsAccount and CheckingAccount?
  • Examples:
    in SavingsAccount : interestRate
    in CheckingAccount : transactionCount
Example: BankAccount class

Example: **BankAccount** constructor:

```java
public BankAccount(double initialAmount, String accountNumber) {
    this.balance = initialAmount;
    this.accountNumber = accountNumber;
}
```

**CheckingAccount** constructor:

```java
public CheckingAccount(double initialAmount, String accountNumber) {
    super(initialAmount, accountNumber);
    transactionCount = 0;
}
```
Example: BankAccount Class

• What new methods might we then have in subclasses SavingsAccount and CheckingAccount?
  • In SavingsAccount:
    • addInterest
    • getInterestRate
  • In CheckingAccount:
    • deductFees
    • different deposit – why?
    • different withdraw – why?
Overriding Methods

• A derived class can define a method with the *same signature* as a method in the parent class
  • The child’s method *overrides* the parent’s method
  • *Example*: methods deposit and withdraw in CheckingAccount override deposit and withdraw of BankAccount
  • *Example*: method toString in Square overrides toString of Rectangle
Overriding Methods

• Which method is actually executed at run time?
  • It depends on *which object is used to invoke the method*
  • *Example:*
    Rectangle r = new Rectangle(4,5);
    Square s = new Square(5);
    System.out.println(r.toString());
    System.out.println(s.toString());

• Note that a method defined with the **final** modifier cannot be overridden
More on the **super** Reference

- Allows us to invoke a method of the parent class that was overridden in the child class
  - **Example:**
    ```java
    public void deposit (double amount) { 
      balance = balance + amount;
    }
    ```
    ```java
    public void deposit (double amount) {
      transactionCount +=
      super.deposit (amount);
    }
    ```

What would happen if we did not have the **super** reference here?
Superclass Variables

- A variable of the *superclass* type may *reference* an object of a *subclass* type
  - *Examples* *(see diagrams next page):*
    
    Square s = new Square(5);
    Rectangle r = s;
    
    Rectangle t = new Square(6);

- A variable of the *subclass* type may *not* reference an object of the *superclass* type
  - Why not?
Superclass Variables

Square s
Rectangle r
Square s1
Rectangle t

Square object
5x5

Rectangle object
6x16
Type of an Object

• Note that the type of an object is determined when it is created, and does not change!

• Examples:

  … = new Rectangle(2,5);
  … = new BankAccount(45.65, “12345”);

• Notice that we are not talking about the type of a variable here
Polymorphism

- **Polymorphism**: the principle that behaviour can vary, depending on the *type of the object* being manipulated

- With inheritance, a *variable* can refer to objects of *different* types during its lifetime

- **Example**:
  
  ```
  Rectangle r;
  r = new Rectangle(2,5);
  System.out.println(r.toString( ));
  ...
  r = new Square(2);
  System.out.println(r.toString( ));
  ```

What’s printed depends on the actual type of the object (*not* the type of the variable)
Polymorphism

• When is it known which method should be invoked? **Not until run time!**
  • This is called **dynamic binding** or **late binding** of the **variable** to the **type of the object**
  • Why is this not known at compile time?

Example:
```java
if ( … )
    r = new Rectangle(2,5);
else
    r = new Square(2);
System.out.println(r.toString( ));
```
Dynamic (Late) Binding

• What happens when a **superclass** variable references an object of a **subclass** type, and a method is invoked on that object?

  **Example:**
  Rectangle r = new Square(5);

• The method *must* exist in the superclass (or one of its ancestors) or there will be a compiler error

  **Example:**
  System.out.println(r.getSide( ));

  **Not legal:** r may not always reference a **Square** object
Dynamic (Late) Binding

• If the method also exists in the subclass, the method from the subclass is invoked (this is **overriding**)

  **Example:** what will be printed by
  ```java
  System.out.println(r.toString( ));
  ```

• If the method does **not** exist in the subclass, the method from the superclass is invoked

  **Example:** is this legal?
  ```java
  System.out.println(r.getWidth( ));
  ```
Casting Reference Variables

• Go back to the example:

```java
Rectangle r = new Square(5);
System.out.println( r.getSide() );
```

• This will generate a compiler error (why?)
• How could we fix it?
  • We can let the compiler know that we *intend* our variable `r` to reference a `Square` object, by *casting* it to type `Square`
Review: Casting Primitive Types

- **Recall**: we have used casting to convert one primitive type to another
  - **Examples**: why are we casting here?

```java
int i, j, n;

n = (int) Math.random();
double q = (double) i / (double) j;
```

- Note that this actually changes the *representation* from integer to double or vice versa
Casting Reference Variables

• We can also cast from one class type to another **within an inheritance hierarchy**

• Fix our previous example by casting:
  Rectangle \( r = \text{new Square}(5); \)
  System.out.println(((Square) r).getSide());

• The **compiler** is now happy with our **intention** that \( r \) references a Square object!
  • We can think of this as doing a *temporary* “type conversion” for the variable
Casting Reference Variables

• But, what if \( r \) did not reference a Square object when casting took place?

    Rectangle \( r = \) new Rectangle(2,5);
    ...
    System.out.println(((Square) r).getSide());

• The compiler is happy, but we would get a \textit{runtime error} (why?)
IndexOf Operator

A safer fix: use the `instanceof` operator

```java
if (r instanceof Square)
{
    System.out.println(((Square)r).getSide());
}
```

- Note that `instanceof` is an `operator`, not a method
- It tests whether the referenced object is an instance of a particular class, and gives the expression the value `true` or `false`
Class Hierarchies

- A derived class can be the parent of classes derived from it
- A single parent class can have many child classes
- **Sibling**: children of the same parent

```
Animal
  
  Reptile
    Snake
    Lizard
  
  Bird
    Parrot
  
  Mammal
    Horse
    Bat
```
Java’s Class Hierarchy

- A class called **Object** is at the top of the class hierarchy so, by default, *any* class extends **Object**
Java’s Class Hierarchy

• Some methods defined in the `Object` class are:
  • `public boolean equals(Object obj);`
  • `public String toString();`

• So, will these methods exist in all classes?
Object methods

- **toString** method: returns a string containing the object’s class name followed by a unique numeric value (the “hash code” of the object, or address that says where it is stored)

- **Example:** Suppose we had *not* defined a **toString** in the Person class. Then the code

  ```java
  Person friend = new Person("Snoopy", "Dog", ";
  System.out.println(friend);
  ``

  would print:

  ```java
  Person@10b62c9
  ```

- Not very meaningful to us, so we usually *override* this method in classes we write
Object methods

• equals method: returns true if the two object references refer to the same object
  • Is this state equivalence or identity equivalence?
  • We often override this method in classes we write, for example if we want equality to mean that the objects hold equal data
Using the Object class

• A variable of type Object can reference an object of any type! (why?)
  • Example:
    Object obj = new Rectangle(5,6);

• So, an array whose elements are of type Object can store any type of object

• It can even store a mix of object types
  • Example:
    Object[] stuff = new Object[10];
    stuff[0] = new Rectangle(5,6);
    stuff[1] = new Integer(25);
    ...

Using the **Object** class

- When an element of the array is obtained, it can be **cast** to its particular (sub)class type, for example:

  ```java
  System.out.println((Rectangle)stuff[0]).area( );
  ```

- We can create a general collection of objects of type **Object**