



# Java Programming

5-1

## Java Class Design – Abstract Classes





# Overview

This lesson covers the following topics:

- Use Abstract Classes
- Use the instanceof operator to compare object types
- Use virtual method invocation
- Use upward and downward casts



# Abstract Classes

- An abstract class provides a base class from which other classes extend.
- Abstract classes can provide:
  - Implemented methods: Those that are fully implemented and their child classes may use them.
  - Abstract methods: Those that have no implementation and require child classes to implement them.
- Abstract classes do not allow construction directly but you can create subclasses or static nested classes to instantiate them.



# Abstract Classes

An abstract class:

- Must be defined by using the abstract keyword.
- Cannot be instantiated into objects.
- Can have local declared variables.
- Can have method definitions and implementations.
- Must be subclassed rather than implemented.



# More Information about Abstract Classes

- If a concrete class uses an abstract class, it must inherit it. Inheritance precludes inheriting from another class.
- Abstract classes are important when all derived classes should share certain methods.
- The alternative is to use an interface, then define at least one concrete class that implements the interface.
- This means you can use the concrete class as an object type for a variable in any number of programs which provides much greater flexibility.



# Abstract Class or Interface

- There is no golden rule on whether to use Interfaces, Abstract classes or both.
- An abstract class usually has a stronger relationship between itself and the classes that will be derived from it than interfaces.
- Classes and Interfaces can implement multiple interfaces. Where a class can only be a sub class of one abstract class.
- Abstract classes allow methods to be defined.

# Bank Account as Abstract

- Previously we had implemented an interface for the Account class.

```
public interface InterfaceBankAccount{  
    public final String bank= "JavaBank";  
    public void deposit(int amt);  
    public void withdraw(int amt);  
    public int getbalance();  
}
```

- We could have used an abstract class instead.



# Bank Account as Abstract

- Interface

```
public interface InterfaceBankAccount
{
    public final String bank= "JavaBank";
    public void deposit(int amt);
    public void withdraw(int amt);
    public int getbalance();
}
```

- Abstract Class

```
public abstract class AbstractBankAccount {
    public final String bank= "JavaBank";
    public abstract void deposit(int amt);
    public abstract void withdraw(int amt);
    public abstract int getbalance();
}
```

# Bank Account as Abstract

- In this implementation there is no advantage of using an abstract class over an interface.
- We would be better using an interface especially if all bank accounts had to implement only those defined methods.
- The interface approach would also allows us to use other interfaces in our class design.
- If we wanted some base functionality to be defined then we would use an abstract class.
- An example is shown on the following slide.

# Bank Account as Abstract

```
public abstract class AbstractBankAccount {
    String accountname;
    int accountnum;
    int balance;
    public void deposit(int amt)
    {
        balance=balance+amt;
    }
    public void withdraw(int amt)
    {
        balance=balance-amt;
    }
    public void setaccountname(String name)
    {
        accountname = name;
    }
    public void setaccountnum(int num)
    {
        accountnum = num;
    }
    public abstract int getbalance();
    public abstract void print();
}
```



# The instanceof Operator

- The instanceof operator allows you to determine the type of an object.
- It takes an object on the left side of the operator and a type on the right side of the operator and returns a boolean value indicating whether the object belongs to that type or not.
- For example:
  - (String instanceof Object) would return true.
  - (String instanceof Integer) would return false.

# Using the instanceof Operator

This could be useful when you desire to have different object types call methods specific to their type.

```
public class CustomerAccounts {  
    private InterfaceBankAccount[] bankaccount = new  
        InterfaceBankAccount[10];  
    public CustomerAccounts() {  
        InterfaceBankAccount[] b = {  
            new Account("Sanjay Gupta",11556,300),  
            new CreditAccount("Sally Walls",66778,1000,500)  
        };  
        this.bankaccount = b;  
    }  
}
```

# instanceof Operator Example

In this example, the instanceof operator is used to find all of the CreditAccounts in an array and add them to a String called list.

```
public String getAllCreditAccounts() {
    String list = "";
    for (int i = 0; i < this.bankaccount.length; i++) {
        if (this.bankaccount[i] instanceof CreditAccount) {
            if (list.length() == 0) {
                list += this.bankaccount[i]; }
            else {
                list += ", " + this.bankaccount[i];
            }
        }
    }
    return list;
}
```

# instanceOf Operator Example Explained

- In the previous instanceof operator example:
  - The array is defined using the InterfaceBankAccount interface. It can hold any class that solely implements its methods and extends them with other specialization methods and variables.
- As the program reads through the array, it assigns the string value from the toString() method to the list variable.
- You should always consider overriding the toString(), hash(), and equals() methods in your user-defined classes.



# Virtual Method Invocation

- Virtual method invocation is the process where Java identifies the type of subclass and calls its implementation of a method.
- When the Java virtual machine invokes a class method, it selects the method to invoke based on the type of the object reference, which is always known at compile time.
- On the other hand, when the virtual machine invokes an instance method, it selects the method to invoke based on the actual class of the object, which may only be known at runtime.



# Virtual Method Invocation Example

```
...
public String toString() {
    String list = "";
    for (int i = 0; i < this.bankaccount.length; i++) {
        if (list.length() == 0) {
            list += this.bankaccount[i]; }
        else {
            list += "\n : " +this.bankaccount[i];}
    }
    return super.toString() + "\n" + list + "\n"
}
...
```

Assigning the object to the String calls that object's toString method.



# Virtual Method Invocation Example Explained

- The `toString()` method uses virtual method invocation by calling the `toString()` method of the different subclasses.
- The decision is made by JVM to call the `toString` method implemented in the subclasses rather than the `toString` method in the superclass or the `Object` class.



# Upcasting and Downcasting

- Casting is the principal of changing the object type during assignment.
- Upcasting loses access to specialized methods in the subclassed object instance.
- Downcasting gains access to specialized methods of the subclass.



# Upcasting and Downcasting Example

- Casting objects works somewhat like casting primitives.
- For example, when downcasting a double to an int, the values to the right of the decimal point are lost.
- This is a loss of precision.
- For that reason, you must explicitly specify the downcast type.
- For example:

```
double complexNumber = 45.75L;  
int simpleNumber = 34;  
  
simpleNumber = (int) complexNumber;
```

# Upcasting and Downcasting Example

Upcasting does not risk the loss of precision nor require you to specify the new data type in parentheses.

# Upcasting Example

- Even though CreditAccount is an Account, it was upcast to Account so it lost access to its Credit Account specific methods and fields.
- Attempts to access subclass methods fail at compile time.

```
...  
Account account =  
    (Account) new CreditAccount("Sally Walls",66778,1000,500);  
...  
    account.getcreditlimit();  
...
```

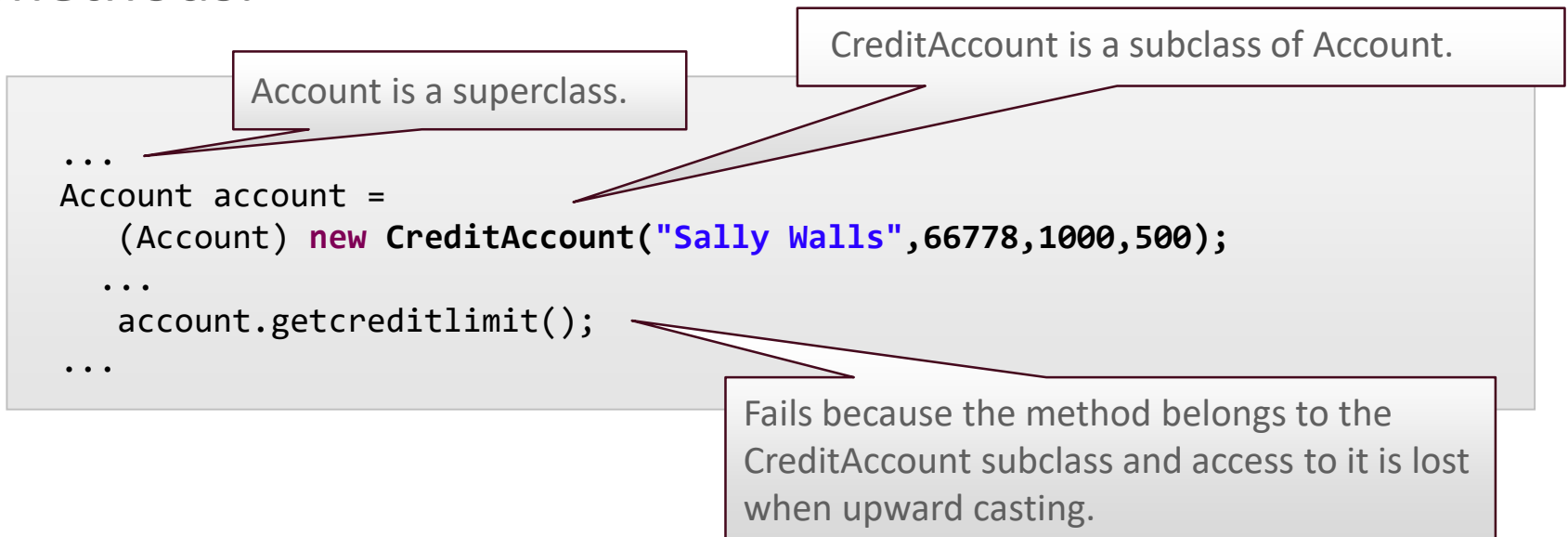
Account is a superclass.

CreditAccount is a subclass of Account.

Fails because the method belongs to the CreditAccount subclass and access to it is lost when upward casting.

# Upcasting Example Explained

- The loss of precision in object assignments differs from primitive data types.
- Upcasting from a specialized subclass to a generalized superclass loses access to the specialized variables and methods.



# Downcasting Example

- Downcasting the superclass works here because the runtime object is actually a CreditAccount instance.
- The Account instance is first cast to a subclass of itself before running the CreditAccount instance method.

```
...  
Account account =  
    (Account) new CreditAccount("Sally Walls",66778,1000,500);  
...  
    ((CreditAccount)account).getcreditlimit();  
...
```

Account is a superclass.

CreditAccount is a subclass of Account.

Succeeds because the method belongs to the CreditAccount subclass.



# Downcasting Example Explained

Downcasting from a generalized to specialized subclass returns access to the specialized variables and methods if they were there from a previous upcast.

```
...  
Account account =  
    (Account) new CreditAccount("Sally Walls",66778,1000,500);  
...  
    ((CreditAccount)account).getcreditlimit();  
...
```

Account is a superclass.

CreditAccount is a subclass of Account.

Succeeds because the method belongs to the CreditAccount subclass.

# Terminology

Key terms used in this lesson included:

- Abstract class
- instanceof
- Casting
- Downward cast
- Interface
- Upward cast
- Virtual method invocation

# Summary

In this lesson, you should have learned how to:

- Model business problems using Java classes
- Use Abstract Classes
- Use the instanceof operator to compare object types
- Use virtual method invocation
- Use upward and downward casts

