
ORACLE
Academy

Java Programming

2-3

Java Class Design – Abstract Classes

ORACLE
Academy

Overview

- This lesson covers the following topics:
  - Use Abstract Classes
  - Use virtual method invocation
  - Use the instanceof operator to compare object types
  - 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 (**extends**) rather than implemented (implements)

Unlike a normal class we cannot create an instance of an abstract class.

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

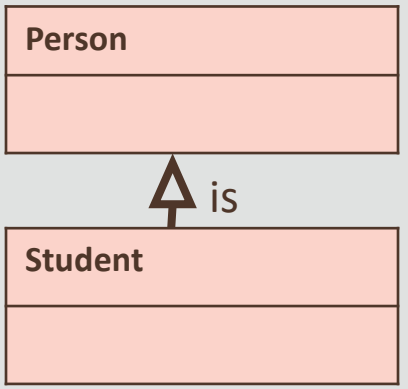
When you subclass from an Abstract class you are forced to implement its abstract methods

Abstract Class or Interface

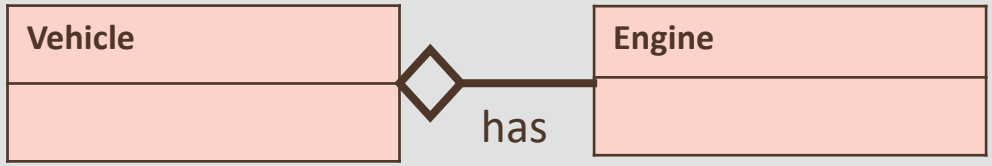
- There is no golden rule on whether to use Interfaces, Abstract classes or both
- An abstract class usually has a stronger relationship between itself and the classes that will be derived from it than interfaces
- Classes and Interfaces can implement multiple interfaces whereas a class can only be a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

Abstract Class or Interface

- Abstract classes often have an " Is-A" relationship
  - A Student is a Person, so Person may be better as an Abstract Class



- Interfaces often have a "Has-A" relationship
  - A Vehicle has an Engine, so Engine may be better as an Interface



Bank Account as Abstract

- Previously an interface for the Account class had been implemented

```
public interface InterfaceBankAccount
{
   public final String BANK= "JavaBank";

   public void deposit(int amt);
   public void withdraw(int amt);
   public int getBalance();
   public String getBankName();

}//end interface InterfaceBankAccount
```

- An abstract class could have been used instead…

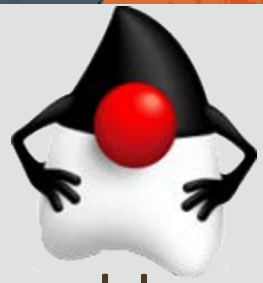
Bank Account as Abstract

- Abstract Class

```
abstract public class AbstractBankAccount {

   public final String BANK= "JavaBank";

   abstract public void deposit(int amt);
   abstract public void withdraw(int amt);
   abstract public int getBalance();
   abstract public String getBankName();

}//end class AbstractBankAccount
```

It is not common to name an Abstract class with the name Abstract. This is done in the above example to help differentiate between the interface and the abstract class.

Abstract Class Task



- **Quick Task:** As an Account is a BankAccount it would be better to use the abstract class

a) Create the AbstractBankAccount abstract class from the previous slide in your JavaBank project

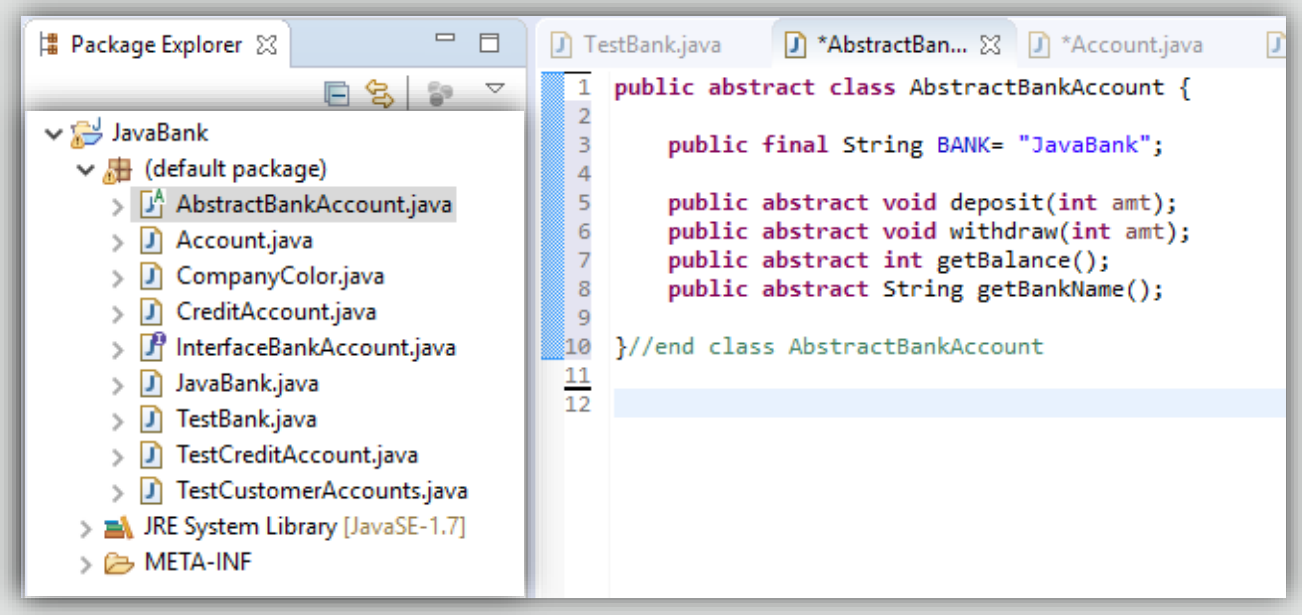
b) Update the Account class definition to extend the abstract class instead of implementing the interface

c) Run and test your code using the TestBank class!

Abstract Class Suggested Solution

- **Quick Task:** As an Account is a BankAccount it would be better to use the abstract class

a) Create the AbstractBankAccount abstract class from the previous slide in your JavaBank project



Abstract Class Suggested Solution

- **Quick Task:** As an Account is a BankAccount it would be better to use the abstract class

b) Update the Account class definition to extend the abstract class instead of implementing the interface

```
public class Account extends AbstractBankAccount{

   // class variables
   protected String accountName;
   protected int accountNum;
   protected int balance;

   //default constructor for Account
   public Account()
   {
```

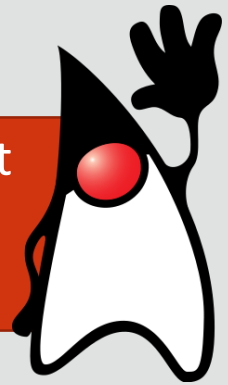
Abstract Class Suggested Solution

- **Quick Task:** As an Account is a BankAccount it would be better to use the abstract class

c) Run and test your code using the TestBank class!

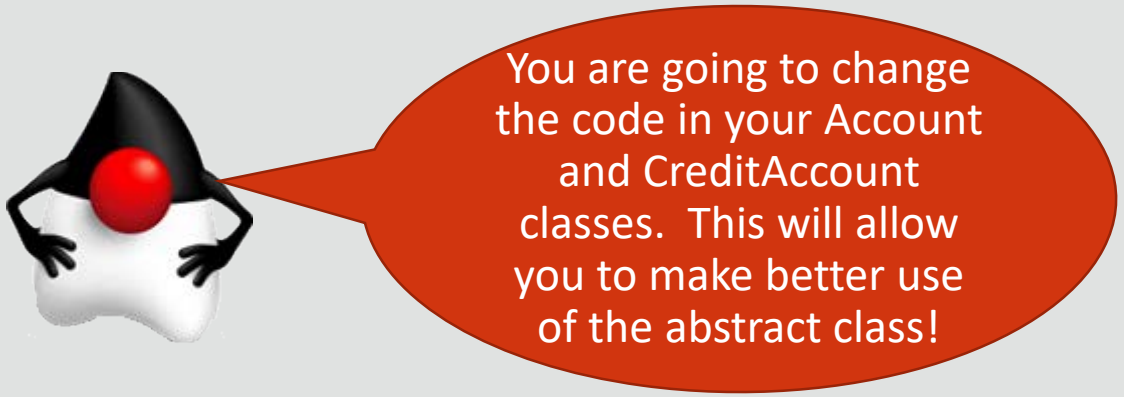
```
Bank Name      : JavaBank
Account Holder : Ilya Mustafana
Account Number : 44559
Account balance: 1000
```

In this implementation there is no advantage of using an abstract class over an interface. Currently both the interface and the abstract class do exactly the same thing.



Bank Account as Abstract

- An interface would be better if all bank accounts had to implement only those defined methods
- The interface approach would allow the use of other interfaces in the class design
- If some base functionality was required to be defined, then an abstract class would be used



Updating the Account/AbstractBankAccount Classes

1. Move the instance fields from the Account class to the AbstractBankAccount class


```
abstract public class AbstractBankAccount {
   //Instance Fields
   public final String BANK= "JavaBank";
   protected String accountName;
   protected int accountNum;
   protected int balance;

   abstract public void deposit(int amt);
   abstract public void withdraw(int amt);
   abstract public int getBalance();
   abstract public String getBankName();
}//end class AbstractBankAccount
```

These are the fields that will be accessed by all subclasses.

Updating the Account/AbstractBankAccount Class

2. Move the relevant getter/setter methods from the Account class to the AbstractBankAccount class
3. Move the withdraw method as this executes the same for both Account and CreditAccount classes

```
   protected int balance;

   abstract public void deposit(int amt);

   public String getBankName(){/*code*/}
   public String getAccountName(){/*code*/}
   public void setAccountName(String name){/*code*/}
   public int getAccountNum(){/*code*/}
   public void setAccountNum(int num){/*code*/}
   public int getBalance(){/*code*/}
   public void setBalance(int num){/*code*/}
   public void withdraw(int amt ){/*code*/}
}//end class AbstractBankAccount
```

Updating the Account Class

4. Delete the default constructor in the Account class so that you cannot create an empty Account object

```
public class Account extends AbstractBankAccount{
   //Instance Fields removed for space on the slide

   //constructor for Account
   public Account(String name, int num, int amt)
   {
      accountName=name;
      accountNum=num;
      balance=amt;
   }//end constructor method

   //make a deposit to the balance
```

Updating the Account/AbstractBankAccount Class

5. Copy the remaining constructor from the Account class into the AbstractBankAccount class Update the constructor name to match the class

```
abstract public class AbstractBankAccount {
   //Instance Fields removed for space on the slide

   //constructor for AbstractBankAccount
   public AbstractBankAccount(String name, int num, int amt)
   {
      accountName=name;
      accountNum=num;
      balance=amt;
   }//end constructor method

   abstract public void deposit(int amt);
```

Updating the Account Class

6. Add a private instance field named bonusValue that can store an integer value in the Account class

```
public class Account extends AbstractBankAccount{

    // Instance Fields
    private int bonusValue;
```

If you open an account you get a bonus value added to your opening balance:

- Between $1 and $100 you get $10
- Between $101 and $300 you get $20
- Over $300 you get $30

TASK: Create a static method named calculateInitialBonusValue that returns an int value that carries out this calculation!

ORACLE Academy

Updating the Account Class

7. Your completed method should look like:

```
}//end constructor method

private static int calculateInitialBonusValue(int amt) {
   if(amt >= 1 && amt <= 100)
      return 10;
   else if(amt <= 300)
      return 20;
   else
      return 30;
   //endif
}//end method calculateInitialBonusValue
```

This method deals with accounts opened with a positive value greater than zero for their initial balance.
Adding error handling for this will come later in the course.

Updating the Account Class

8. Update the Account constructor to call the super constructor in AbstractBankAccount
   - Use the **name**, **num** and the result of **amt** + **calculateInitialBonusValue** as the arguments

```
public class Account extends AbstractBankAccount{

   // Instance Fields
   private int bonusValue;

   //constructor for Account
   Account(String name, int num, int amt)
   {
      super(name, num, (amt + calculateInitialBonusValue(amt)));
   }//end constructor method

   private static int calculateInitialBonusValue(int amt) {
```

Updating the Account Class

9. Use the **calculateInitialBonusValue** method to assign the initial value to the private bonusValue field

```
public class Account extends AbstractBankAccount{

   // Instance Fields
   private int bonusValue;

   //constructor for Account
   Account(String name, int num, int amt)
   {
      super(name, num, (amt + calculateInitialBonusValue(amt)));
      bonusValue = calculateInitialBonusValue(amt);
   }//end constructor method

   private static int calculateInitialBonusValue(int amt) {
```

Updating the Account Class

10. If you have an Account that is not a credit account, you will receive an additional 10% of your initial bonus value added to each deposit over $100

```
}//end method calculateInitialBonusValue

//make a deposit to the balance
public void deposit(int amt)
{
   if(amt>100)
      balance=balance+(amt + (int)(bonusValue * 0.1));
   else
      balance=balance+amt;
   //endif
}//end method deposit
```

This additional payment is unique to this account type. A credit account does not have this feature.

Updating the CreditAccount Class

11. Have the CreditAccount class inherit from AbstractBankAccount instead of Account

```
public class CreditAccount extends AbstractBankAccount{

   private int creditLimit;

   //default constructor for CreditAccount
   CreditAccount()
```

The AbstractBankAccount class is the superclass to both the Account and CreditAccount classes. They can access the implemented methods at the super level and will have to implement the abstract methods at the sub level!

Updating the CreditAccount Class

- When you extend the abstract class you are required to add the unimplemented methods

12. For the AbstractBankAccount class this means that you have to implement a deposit method that accepts an integer parameter and adds it to the current value of the balance

```
    @Override
    public void deposit(int amt) {
        balance=balance+amt;
    }//end method deposit

}//end class CreditAccount
```

Updating the CreditAccount Class

13. Update the default constructor to accept parameters for the name, number and initial amount for the credit account
    - This allows the creation of a credit account with the default $100 credit limit

```
//default constructor for CreditAccount
CreditAccount(String name, int num, int amt)
{
   super(name,num,amt);
   this.creditLimit=100;
}//end constructor method
```

The super call must be the first line in the constructor and the arguments must match the parameter list in the super constructor.

Updating the CreditAccount Class

14. A constructor for a basic credit account already exists where a $100 dollar credit limit is assigned

- JavaBank wants to implement a new scheme to make the processing of credit accounts easier:

If you open a credit account you get a better credit limit depending on the size of your initial deposit:

- Between $1 and $2000 you get $100
- Between $2001 and $4000 you get $200
- Over $4000 you get $300

TASK: Create a static method named calculateCreditLimit that returns an int value that carries out this calculation!

ORACLE Academy

Updating the Account Class

15. Your completed method should look like:

```
}//end constructor method

private static int calculateCreditLimit(int amt) {
   if(amt>1 && amt<=2000)
      return 100;
   else if(amt<=4000)
      return 200;
   else
      return 300;
   //endif
}//end method calculateCreditLimit
```

This method has to be static as it is being called before the object has been fully created. Static fields and methods are created first!

Updating the CreditAccount Class

16. Update the default constructor to call calculateCreditLimit() to assign a value for the creditLimit instance field

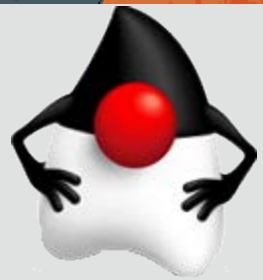
```
public class CreditAccount extends AbstractBankAccount{

   // Instance Fields
   private int creditLimit;

   //default constructor for CreditAccount
   CreditAccount(String name, int num, int amt)
   {
      super(name,num,amt);
      this.creditLimit=calculateCreditLimit(amt);
   }//end constructor method
```

Updating the CreditAccount Class

17. Update the print method in the CreditAccount class so that the output resembles that of the Account class but also displays the credit limit

```
//print method
public void print()
{
   System.out.println("\nBank Name      : " + getBankName() +
                      "\nAccount Holder : " + accountName +
                      "\nAccount Number : " + accountNum +
                      "\nAccount balance: " + balance +
                      "\nCredit Limit   : " + creditLimit);
}//end method print
```

Testing the Abstract Class Task 1



- **Quick Task:** You now need to test that your changes worked with your test classes

a) Update the **TestBank.java** class so that the values for the a2 and a3 accounts are provided through the constructor instead of through the setter method calls

b) Remove the setter method calls from the TestBank class!

c) Run and test your code using the TestBank class!

Testing the Abstract Class Solution 1

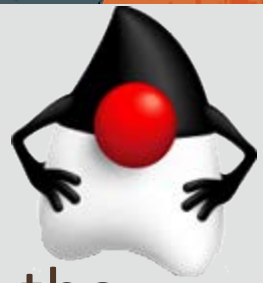
```
public class TestBank {
   public static void main(String[] args) {

      // Instantiate 3 accounts
      // Using constructor with values
      Account a1 = new Account("Sanjay Gupta",11556,300);
      Account a2 = new Account("He Xai",22338,500);
      Account a3 = new Account("Ilya Mustafana",44559,1000);

      // Print accounts
      a1.print();
      a2.print();
      a3.print();

   }//end method main
}//end class testBank
```

Testing the Abstract Class Task 1

- Your output should look like this:

```
Bank Name       : JavaBank
Account Holder : Sanjay Gupta
Account Number : 11556
Account balance: 320

Bank Name       : JavaBank
Account Holder : He Xai
Account Number : 22338
Account balance: 530

Bank Name       : JavaBank
Account Holder : Ilya Mustafana
Account Number : 44559
Account balance: 1030
```

Testing the Abstract Class Task 2



- **Quick Task:** Test that your changes worked with the TestCreditAccount class

a) Update the **TestCreditAccount.java** class so that the values for the a2 and a3 accounts are provided through the constructor instead of through the setter method calls

b) Update the instantiation of the c1 account so the values are provided through the constructor instead of through the setter method calls

c) Account is no longer the superclass so update the instantiation code to create a creditAccount object

d) Remove the setter calls and run and test your code

Testing the Abstract Class Solution 2

```
public class TestCreditAccount {
   public static void main(String[] args) {
      // Instantiate 3 accounts using constructor with values
      Account a1 = new Account("Sanjay Gupta",11556,300);
      Account a2 = new Account("He Xai",22338,500);
      Account a3 = new Account("Ilya Mustafana",44559,1000);

      // Instantiate 2 credit accounts using constructor with
      // values which will call constructor from super
      CreditAccount c1 = new CreditAccount("A.N Other", 88776, 500);
      CreditAccount c2 = new CreditAccount("Another",66778,1000,500);

      // Print accounts
      a1.print();
      a2.print();
      a3.print();
      c1.print();
      c2.print();
   }//end method main
}//end class testCreditAccount
```

Testing the Abstract Class Task 1

- Your output should look like this:

```
Bank Name      : JavaBank
Account Holder : Sanjay Gupta
Account Number : 11556
Account balance: 320

Bank Name      : JavaBank
Account Holder : He Xai
Account Number : 22338
Account balance: 530

Bank Name      : JavaBank
Account Holder : Ilya Mustafana
Account Number : 44559
Account balance: 1030
```

```
Bank Name      : JavaBank
Account Holder : A.N Other
Account Number : 88776
Account balance: 500
Credit Limit   : 100

Bank Name      : JavaBank
Account Holder : Another
Account Number : 66778
Account balance: 1000
Credit Limit   : 500
```

Virtual Method Invocation

- Virtual method invocation is the process where Java identifies the type of subclass and calls that class' 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 Explained

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

Remember we can still access the parent's toString method by calling super.toString().

Testing the Abstract Class Task 1

- **Quick Task:** Create toString() methods for your bank application

a) Create a toString() method for the AbstractBankAccount class that will return a String value that matches the current output of the Account class' print method

b) Create a toString() method that includes a super.toString() method call as well as an output message for the creditLimit instance field in the CreditAccount class

c) Update the testCreditAccount class' print() method calls to use the toString methods instead

Testing the Abstract Class Solution 1

a) AbstractBankAccount class' toString() method

```
    @Override
    public String toString()
    {
        return "\n\nBank Name      : " + getBankName() +
               "\nAccount Holder : " + accountName +
               "\nAccount Number : " + accountNum +
               "\nAccount balance: " + balance;
    }//end method toString

}//end class AbstractBankAccount
```

Testing the Abstract Class Solution 1

b) CreditAccount class' toString() method

```
    @Override
    public String toString() {
        return super.toString() +
                "\nCredit Limit   : " + creditLimit;

    }//end method toString

}//end class CreditAccount
```

Testing the Abstract Class Solution 1

c) Updated testCreditAccount class' toString method calls

```
        // Print accounts
        //a1.print();
        System.out.println(a1);
        //a2.print();
        System.out.println(a2);
        //a3.print();
        System.out.println(a3);
        //c1.print();
        System.out.println(c1);
        //c2.print();
        System.out.println(c2);
    }//end method main
}//end class testCreditAccount
```

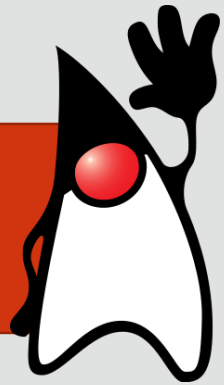
The account objects call the toString() method of the AbstractBankAccount class.

The credit account objects call the toString() method of theCreditAccount class.

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.



The instanceof Operator

- Currently in the test classes (TestBank, TestCreditAccount) the objects are being created as separate instances
- An array remember can only hold values of a single type. Because of this the array would need to be created using the AbstractBankAccount as its type
- The instanceof operator could then be used to identify the exact account type held in the array

An example of this will be shown over the next few slides.
Add it to your JavaBank application.

instanceof Operator Example

1. Create a TestCustomerAccounts class that has a main method
2. Define an array using the InterfaceBankAccount abstract class that can store 5 objects

```
public class TestCustomerAccounts {

   public static void main(String[] args) {

      AbstractBankAccount[] bankAccount = new
AbstractBankAccount[5];

   }//end method main

}//end class TestCustomerAccounts
```

instanceof Operator Example

3. Add these accounts into the bankAccount array

```
public static void main(String[] args) {
   AbstractBankAccount[] bankAccount = new AbstractBankAccount[5];

   // Instantiate 2 credit accounts using constructor with
   bankAccount[0] = new Account("Sanjay Gupta",11556,300);
   bankAccount[1] = new Account("He Xai",22338,500);
   bankAccount[2] = new Account("Ilya Mustafana",44559,1000);

   // Instantiate 2 credit accounts using constructor with
   bankAccount[3] = new CreditAccount("A.N Other", 88776, 500);
   bankAccount[4] = new CreditAccount("Another",66778,1000,500);

}//end method main
```

The bankAccount array can hold any objects that have been created as a subclass of the AbstractBankAccount type!

instanceof Operator Example

4. Create a showAllCustomerAccounts method that accepts the bankAccount array as a parameter and uses an enhanced for loop to display all the accounts

```
   showAllCustomerAccounts(bankAccount);
}//end method main

public static void showAllCustomerAccounts(AbstractBankAccount[]
                                                    bankAccount) {
   System.out.print("\nAll Customer Accounts******");
   for(AbstractBankAccount act:  bankAccount)
      System.out.println(act);
   //endfor
}//end method showAllCustomerAccounts
```

The correct toString() method will be called to display the contents of the object to the console!

instanceof Operator Example

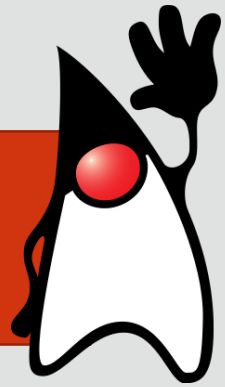
5. Create a showAllAccounts method that is based on the showAllCustomerAccounts method
   - Use an if statement in the enhanced for loop to include the instanceof operator to only show accounts of type Account
   - Call the method from main.

```
   showAllAccounts(bankAccount);
}//end method main

public static void showAllAccounts(AbstractBankAccount[] bankAccount) {
   System.out.print("\nAll Account types******");
   for(AbstractBankAccount act:  bankAccount)
      if (act instanceof Account)
         System.out.println(act);
   //endfor
}//end method getAllAccounts
```

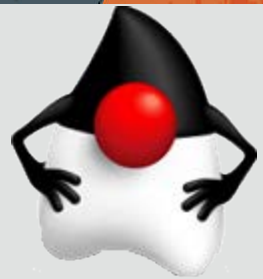
instanceOf Operator Example Explained

- In the previous instanceof operator example:
  - The array is defined using the InterfaceBankAccount abstract class. It can hold any class that implements its methods and extends them with other specialization methods and variables
- As the program reads through the array, it displays the string value from the toString() method to the console

You would normally use the instanceof operator when you have a reference or parameter to an object that is of a super class or interface type and need to know which type of object it refers to



Testing the instanceof operator Task



- **Quick Task:** Identify the and display the different types of bank accounts held in the system

a) Create a showAllCreditAccounts method that is based on the showAllAccounts method

b) Update the if statement to only show the credit accounts to the console

c) In main call the show methods so that all customer accounts, all accounts and then all credit accounts are displayed to the console

instanceof operator Solution

a) and b). ShowAllCreditAccounts method.

```
public static void showAllCreditAccounts(AbstractBankAccount[]
                                                    bankAccount) {
   System.out.print("\nAll Credit Account types******");
   for(AbstractBankAccount act:  bankAccount)
      if (act instanceof CreditAccount)
         System.out.println(act);
      //endif
   //endfor
}//end method getAllCreditAccounts
```

c) Method calls in main

```
   showAllCustomerAccounts(bankAccount);
   showAllAccounts(bankAccount);
   showAllCreditAccounts(bankAccount);
}//end method main
```

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
- Casting objects works somewhat like casting primitives

You may have used casting before on primitives but less so on objects.

Upcasting and Downcasting Example

- **Upcasting** does not risk the loss of precision nor require you to specify the new data type
- This is when going from a smaller to a larger data type so there is no risk of losing information
- Java will implicitly carry out the conversion for you

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

complexNumber = simpleNumber;
```

The complexNumber would become 34 with no loss of precision as a smaller data type (int) has been stored in a larger one (double).

Upcasting and Downcasting Example

- **Downcasting** a double to an int means 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, Java will not allow this implicitly

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

simpleNumber = (int) complexNumber;
```

The simpleNumber would become 45 as we have lost the precision of a double type when storing it in an integer.

Upcasting Example

- Even though CreditAccount is an AbstractBankAccount, here it is upcast to AbstractBankAccount so it has lost access to its Credit Account specific methods and fields
- Accessing subclass methods would fail at compile time

AbstractBankAccount is a superclass.

CreditAccount is a subclass of Account.

```
...

AbstractBankAccount credAct1 = (AbstractBankAccount) new
                    CreditAccount("Ilya Mustafana",44559,1000);

credAct1.getCreditLimit();
...
```

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

Upcasting Example Explained

- Upcasting does not get used that often
- 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
  - One example of when upcasting may be necessary is when you have overloaded methods and do not want to call the specialized one

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

AbstractBankAccount is a superclass.

CreditAccount is a subclass of Account.

```
...
AbstractBankAccount credAct1 = (AbstractBankAccount) new
                    CreditAccount("Ilya Mustafana",44559,1000);

((CreditAccount)credAct1).getCreditLimit();
...
```

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 the object had been previously upcast
- In the example on the previous slide it would return access to the credit specific information held in the JavaBank

Terminology

- Key terms used in this lesson included:
  - Interface
  - Abstract class
  - Virtual method invocation
  - instanceof
  - Casting
  - Upward cast
  - Downward cast

Summary

- In this lesson, you should have learned how to:
  - Use Abstract Classes
  - Use virtual method invocation
  - Use the instanceof operator to compare object types
  - Use upward and downward casts


ORACLE
Academy