CN208 Introduction to Computer Programming

Lecture #9

Interfaces & Array Lists

Pimarn Apipattanamontre

Email: ***pimarn@pimarn.com***

Review for Lecture #8

- Java allows an object reference of the subclass to be stored to that of the superclass.
- Polymorphism is the the principle that the actual type of the object reference determines the method to be called.
- Abstract classes
  - The mechanism that forces programmers to create subclasses and implement some abstract methods.
- Final classes & final methods:
  - The mechanisms that prevent programmers from defining subclasses and overriding methods, respectively.
- Packages & organizing related classes into packages.

Review for Polymorphism

- Why would we want to store an object reference of the subclass to an object reference of the superclass?

```
public class BankAccount
{
  . . .
  public void deposit(double amount) { . . . }
  public void withdraw(double amount) { . . . }
  public double getBalance() { . . . }
  public void transfer(BankAccount other, double amount)
  {
    withdraw(amount);
    other.deposit(amount);
  }

}
```

```
SavingsAccount momsSavings = new SavingsAccount(10000, 0.5);
CheckingAccount somsaksChecking = new CheckingAccount(100);

momsSavings.transfer(somsaksChecking, 1000);
```

- Which *deposit( )* method is used in *transfer( )* method? (the one defined in *BankAccount* or the one redefined in *CheckingAccount*?
  - Method calls are determined by the actual type of the object reference.

Review for Abstract Classes

```
public abstract class BankAccount
{
   public abstract void deductFees();
   . . .
}
```

- To prevent from being forgotten to implement, the *deductFees()* method is declared as an *abstract* method.
- Being an *abstract* method, it forces the implementors of subclasses to define the concrete implementations of this method.
- A class defines an abstract method or that inherits an abstract method without overriding it, must be declared as *abstract*.

Final Classes

- We can use the keyword **final** to prevent programmers from creating subclasses (usually to prevent the occurrence of polymorphism).

```
public final class BankAccount { . . . }
```

Final Methods

```
public class BankAccount
{  . . .
   public boolean verifyPIN(String pin) { . . . }
}
public class SavingsAccount extends BankAccount
{  . . .
   public boolean verifyPIN(String pin) { return true; }
}
```

⇦ Security of any *SavingsAcccount* object is violated!

- We can also use the keyword **final** to prevent programmers from or overriding certain methods.

```
public class BankAccount
{  . . .
   public final boolean verifyPIN(String pin) { . . . }
}
```

The BankAccount Class

```
public class BankAccount
{
  // instance variable
  private double balance;

  // constructors
  public BankAccount(double initBalance) { balance = initBalance; }
  public BankAccount() { balance = 0; }

  // methods
  public double getBalance() { return balance; }
  public void deposit(double amount) { balance = balance + amount; }
  public void withdraw(double amount) { balance = balance - amount; }
}
```

The Triangle Class

```
public class Triangle
{
  // instance variables
  private double height;
  private double width;

  // constructor
  public Triangle(double altitude, double base)
  { height = altitude; width = base; }
  public Triangle() { height = 1.0; width = 1.0; }

  // methods
  public double getArea() { return (0.5 * height * width); }
}
```

The DataSet Class

```
public class DataSet
{
  // instance variable
  private double sum;
  private double count;
  private double max;

  // constructors
  public DataSet()
  { sum = 0; count = 0; max = 0; }

  // methods
  public double getMax() { return max; }
  public void add(double x)
  {
    sum = sum + x;
    if ((count == 0) || (max < x))
      max = x;
    count++;
  }
}
```

```
public class DataSetBankAccount
{
  // instance variable
  private double sum;
  private double count;
  private BankAccount max;

  // constructors
  public DataSetBankAccount()
  { sum = 0; count = 0; max = null; }

  // methods
  public BankAccount getMax() { return max; }
  public void add(BankAccount x)
  {
    sum = sum + x.getBalance();
    if ((count == 0) ||
        (max.getBalance() < x.getBalance()))
      max = x;
    count++;
  }
}
```

The DataSet Class

```
public class DataSet
{
  // instance variable
  private double sum;
  private double count;
  private double max;

  // constructors
  public DataSet()
  { sum = 0; count = 0; max = 0; }

  // methods
  public double getMax() { return max; }
  public void add(double x)
  {
    sum = sum + x;
    if ((count == 0) || (max < x))
      max = x;
    count++;
  }
}
```

```
public class DataSetTriangle
{
  // instance variable
  private double sum;
  private double count;
  private Triangle max;

  // constructors
  public DataSetTriangle()
  { sum = 0; count = 0; max = null; }

  // methods
  public Triangle getMax() { return max; }
  public void add(Triangle x)
  {
    sum = sum + x.getArea();
    if ((count == 0) ||
        (max.getArea() < x.getArea()))
      max = x;
    count++;
  }
}
```

The DataSet Class

```
public class DataSet
{
  // instance variable
  private double sum;
  private double count;
  private MeasurableObject max;

  // constructors
  public DataSet()
  { sum = 0; count = 0; max = null; }

  // methods
  public MeasurableObject getMax()
  { return max; }
  public void add(MeasurableObject x)
  {
    sum = sum + x.getValue();
    if ((count == 0) ||
        (max.getValue() < x.getValue()))
      max = x;
    count++;
  }
}
```

```
public class DataSetTriangle
{
  // instance variable
  private double sum;
  private double count;
  private Triangle max;

  // constructors
  public DataSet()
  { sum = 0; count = 0; max = null; }

  // methods
  public Triangle getMax() { return max; }
  public void add(Triangle x)
  {
    sum = sum + x.getArea();
    if ((count == 0) ||
        (max.getArea() < x.getArea()))
      max = x;
    count++;
  }
}
```

The MeasurableObject Interface

File DataSet.java

```
public class DataSet
{
  // instance variable
  private double sum;
  private double count;
  private MeasurableObject max;

  // constructors
  public DataSet()
  { sum = 0; count = 0; max = null; }

  // methods
  public MeasurableObject getMax()
  { return max; }
  public void add(MeasurableObject x)
  {
    sum = sum + x.getValue();
    if ((count == 0) ||
        (max.getValue() < x.getValue()))
      max = x;
    count++;
  }
}
```

File MeasurableObject.java

```
public interface MeasurableObject
{
  double getValue(); // automatically public
}
```

- A Java *interface* defines a set of methods but no implementation (All methods of the interface are abstract).
- All methods in an *interface* are automatically public.
- Similar to an abstract class, we cannot construct an object of the *interface* type but we can have object reference (object variable) whose type is an *interface*.

Implement the MeasurableObject Interface

```
public class BankAccount implements MeasureableObject
{
  // instance variable
  private double balance;

  // constructors
  public BankAccount(double initBalance)
  { balance = initBalance; }
  public BankAccount() { balance = 0; }

  // methods
  public double getValue() { return balance; }
  public double getBalance() { return balance; }
  public void deposit(double amount)
  { balance = balance + amount; }
  public void withdraw(double amount)
  { balance = balance - amount; }

}
```

⇦ Must specify the keyword public

```
public interface MeasurableObject
{
  double getValue(); // automatically public
}
```

- To realize an *interface*, a class must declare the *interface name* after the **implements** keyword and it must implement all methods defined in the interface.
- A class can realize more than one interface by putting all interface names after the **implements** keyword and separating them with commas. In such a case, the class must then implement all the methods defined in all interfaces.
- Although all methods defined in an interface are automatically public, they must be explicitly declared as public when they are implemented in a class.

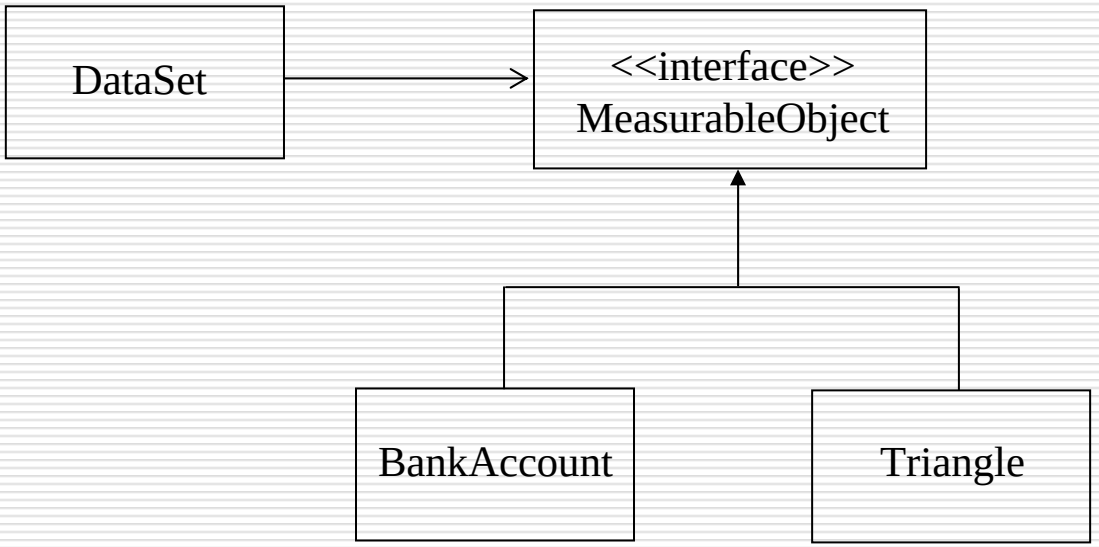
Implement the MeasurableObject Interface

```
public class Triangle implements MeasureableObject
{
  // instance variables
  private double height;
  private double width;

  // constructor
  public Triangle(double altitude, double base)
  { height = altitude; width = base; }
  public Triangle() { height = 1.0; width = 1.0; }

  // methods
  public double getValue()
  { return (0.5 * height * width); }
  public double getArea()
  { return (0.5 * height * width); }

}
```

⇦ Must specify the keyword public

```
public interface MeasurableObject
{
  double getValue(); // automatically public
}
```

The UML Notation



⟶ indicates that the *DataSet* class depends on the *MeasurableObject* interface

⟶ indicates that the *BankAccount* and *Triangle* Classes realize the *MeasurableObject* interface.

Using the DataSet class

```
public class DataSet
{
  // instance variable
  private double sum;
  private double count;
  private MeasurableObject max;

  // constructors
  public DataSet()
  { sum = 0; count = 0; max = null; }

  // methods
  public MeasurableObject getMax()
  { return max; }
  public void add(MeasurableObject x)
  {
    sum = sum + x.getValue();
    if ((count == 0) ||
        (max.getValue() < x.getValue()))
      max = x;
    count++;
  }
}
```

```
public interface MeasurableObject
{
  double getValue(); // automatically public
}


public class DataSetTest1
{
  public static void main(String[] args)
  {
    DataSet bankAccSet = new DataSet();

    bankAccSet.add(new BankAccount());
    bankAccSet.add(new BankAccount(10000));
    bankAccSet.add(new BankAccount(2000));
    MeasurableObject max = bankAccSet.getMax();

    System.out.println("Max balance = " +
                       max.getValue());
  }
}
```

- Polymorphism is applied to the *getValue( )* method, when the *add( )* method is called.
- We can convert from a class type to an interface type (in this case, it happens at the parameter of the *add( )* method) if the class implements the interface.
- For the *bankAccSet* object, the *getValue( )* method implemented in the *BankAccount* class is used.

Using the DataSet class

```
public class DataSet
{
  // instance variable
  private double sum;
  private double count;
  private MeasurableObject max;

  // constructors
  public DataSet()
  { sum = 0; count = 0; max = null; }

  // methods
  public MeasurableObject getMax()
  { return max; }
  public void add(MeasurableObject x)
  {
    sum = sum + x.getValue();
    if ((count == 0) ||
        (max.getValue() < x.getValue()))
      max = x;
    count++;
  }
}
```

```
public interface MeasurableObject
{
  double getValue(); // automatically public
}

public class DataSetTest2
{
  public static void main(String[] args)
  {
    DataSet triangleSet = new DataSet();

    triangleSet.add(new Triangle());
    triangleSet.add(new Triangle(10,20));
    triangleSet.add(new Triangle(20,30));
    MeasurableObject max = triangleSet.getMax();

    System.out.println("Max balance = " +
                       max.getValue());
  }
}
```

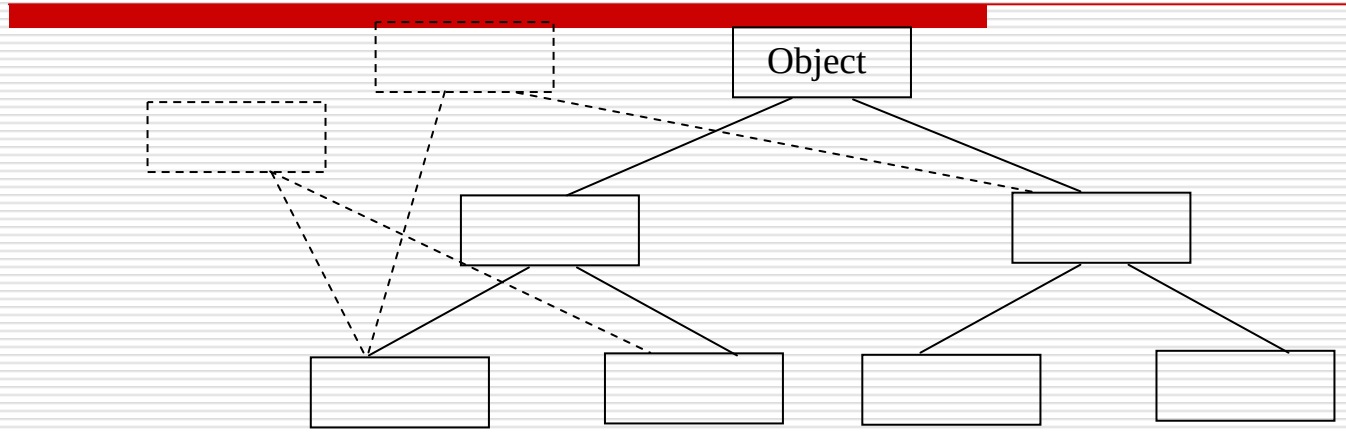
For the *triangleSet* object, the *getValue()* method implemented in the *Triangle* class is used.

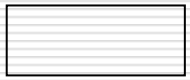
Constants in Interfaces

```
public interface SwingConstants
{
   int NORTH = 1;  // we can omit ‘public static final’
   . . .
}
```

- Interfaces cannot have variables but they can have constants.
- All constants in an interface are automatically ‘*public static final*’ and we usually do not specify all these keywords.
- Since all the constants of an interface must be static, they are stored in the memory belonging to the interface. They are initialized when the interface is loaded, which happens when any of the constants are read for the first time.

Classes and Interfaces





Abstract or Concrete Class

Interface

- Interface are defined outside the class hierarchy while all classes (both abstract and concrete) are in the same class hierarchy since they are inherited from the *Object* class.
- An interface can be implemented by any class regardless of its position in the class hierarchy.
- A class can implement several interfaces.

The Coin Class

```
public class Coin
{
  // instance variables
  double coinValue;
  String coinType;

  // constructor
  public Coin(double aValue, String aName) { coinValue = aValue; coinType = aName; }
  // method
  public double getCoinValue() { return coinValue; }
  public String getCoinType() { return coinType; }
  public boolean equals(Coin aCoin)
  { return ((coinValue == aCoin.getCoinValue()) &&
            (coinType.equals(aCoin.getCoinType()))); }
}
```

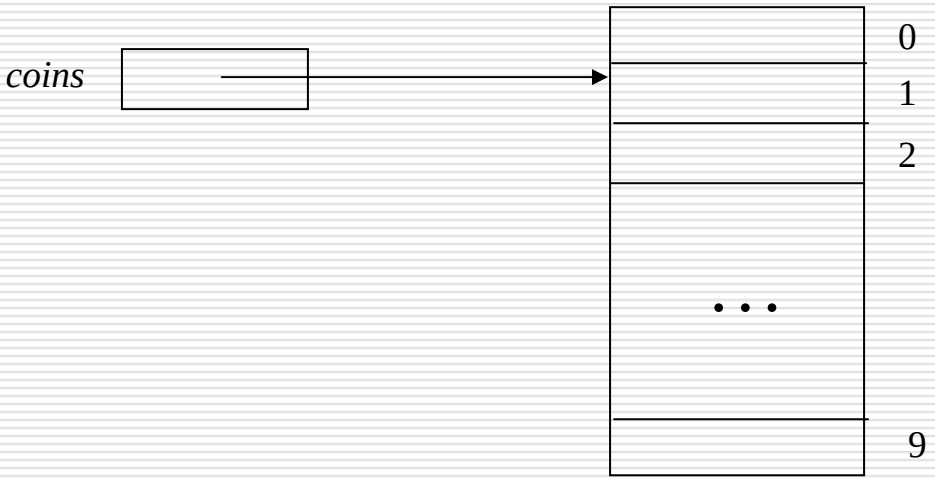
Coin aTenBahtCoin = new Coin(10, “TenBaht”);
Coin aFiveBahtCoin = new Coin(5, “FiveBaht”);
Coin aOneBahtCoin = new Coin(1, “OneBaht”);

The ArrayList Class

- The *ArrayList* class is a Standard Java class defined in the *java.util* package.
- An *array list* is a sequence of objects.
- To create an array list,

```
ArrayList  coins = new ArrayList();
```

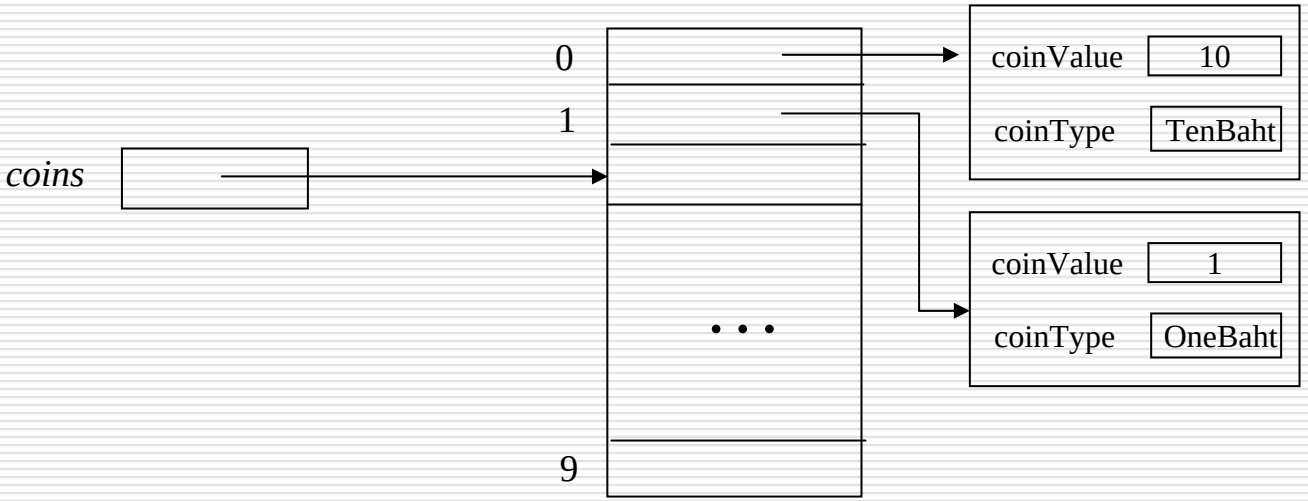
- Construct an *empty* array list with a default capacity of ten object references in the *Object* type.



The ArrayList Class

- We can add an element (an object) to the end of the list of the array by using the *add( )* method defined in the *ArrayList* class.

```
coins.add(new Coin(10.0, "TenBaht"));
coins.add(new Coin(1.0, "OneBaht"));
```



- Each object in the array list has an integer position number, called the *index*.

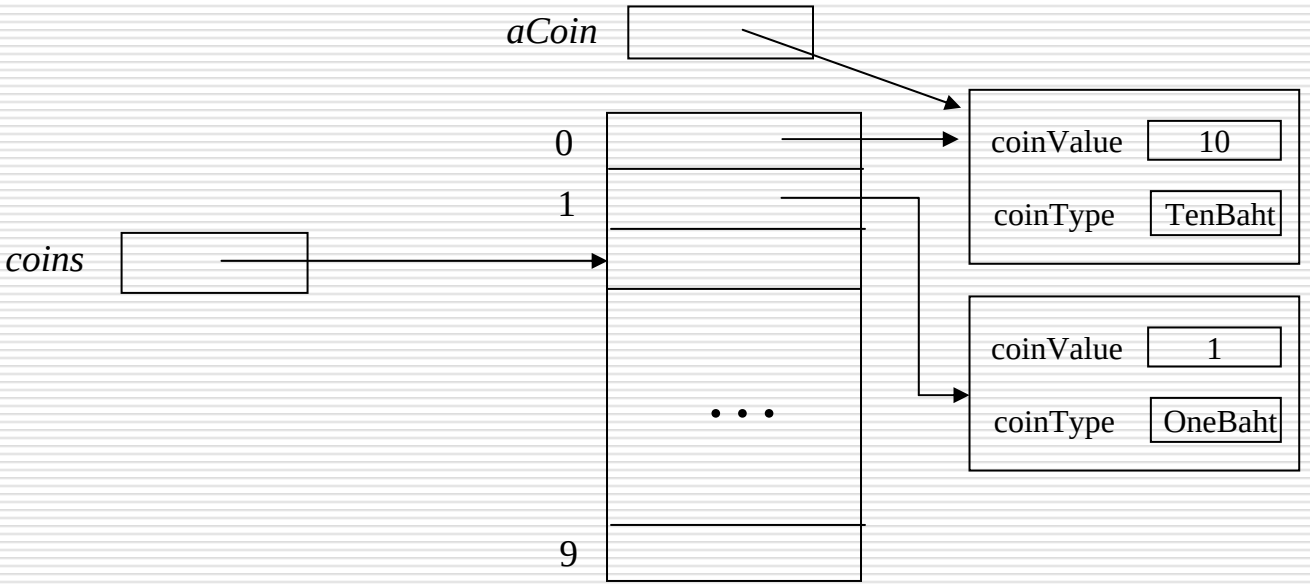
The ArrayList Class

- To get the number of elements of an array list, use the *size( )* method.

```
int i = coins.size(); // i is equal to 2.
```

- The index of the array list ranges from 0 to *size( )* -1 .
- To get an object out of the array list, use the *get( )* method.

```
Coin aCoin = (Coin) coins.get(0);
```

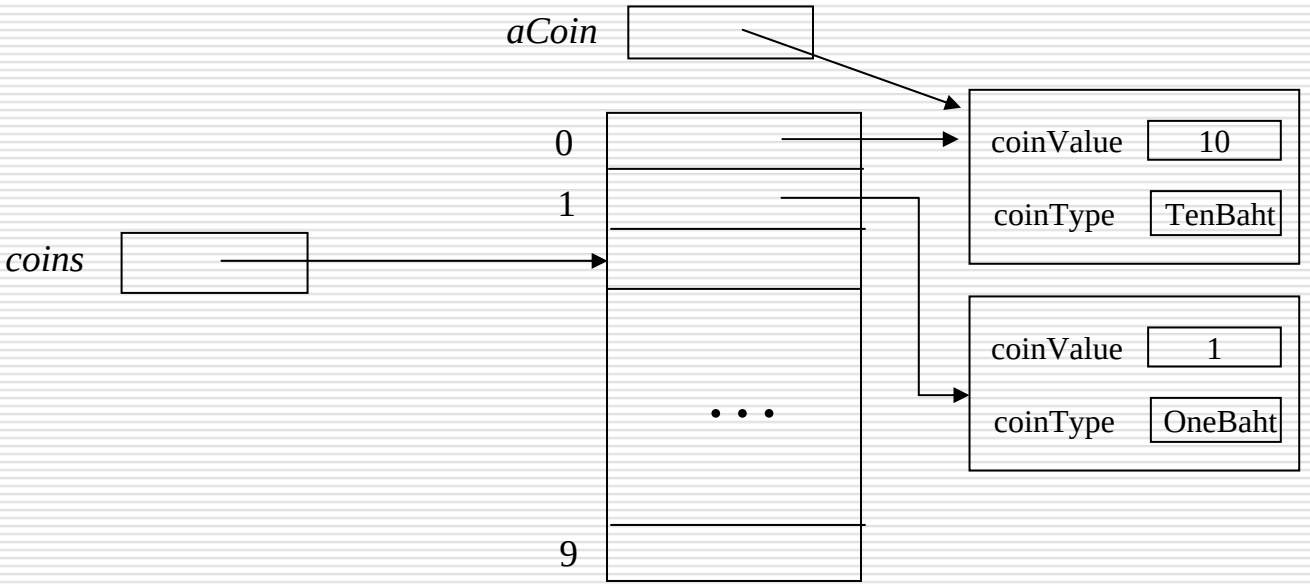


The ArrayList Class

- To get an object out of the array list, use the *get( )* method.

```
Coin aCoin = (Coin) coins.get(0);
```

- When getting an object from the array list, we need to cast the return value of the *get ()* method. This is because the *get()* method defined in the *ArrayList* class has the reference of the *Object* type as its returned value.



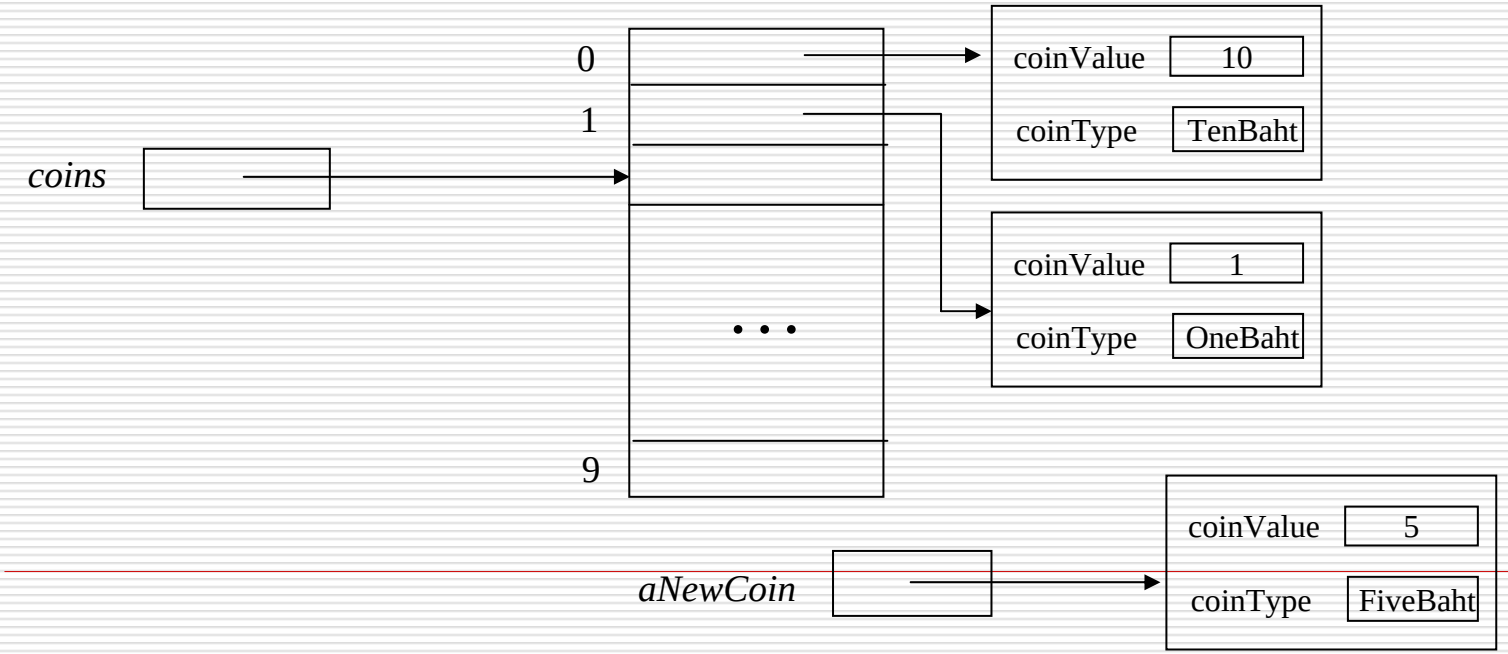
The ArrayList Class

- Accessing a nonexistent position causes an exception to be thrown.

```
Coin aCoin = (Coin) coins.get(4); // throws IndexOutOfBoundsException
```

- To set an array list element to a new value, use the *set( )* method of the ArrayList class.

⇨
```
Coin aNewCoin = (Coin) new Coin(5.0, "FiveBaht"));
coins.set(1, aNewCoin); // set position 1 of the coins array list to aNewCoin
```



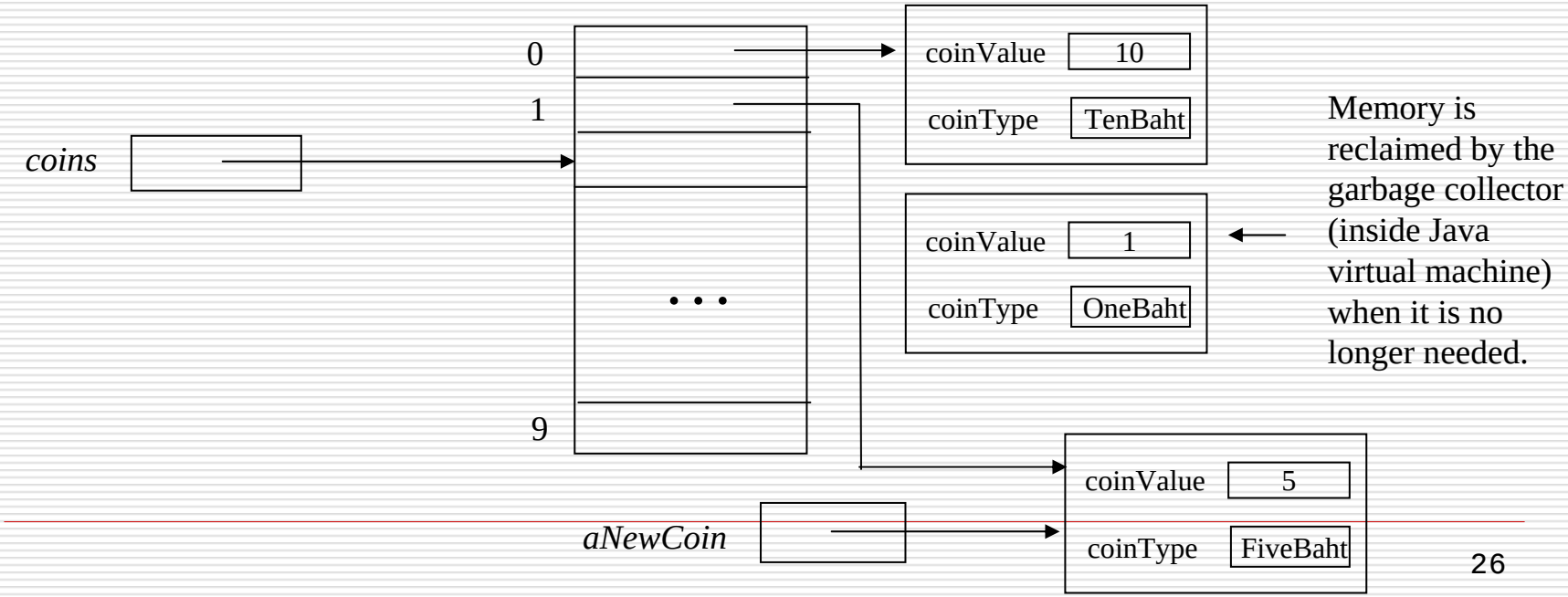
The ArrayList Class

- Accessing a nonexistent position causes an exception to be thrown.

```
Coin aCoin = (Coin) coins.get(4); // throws IndexOutOfBoundsException
```

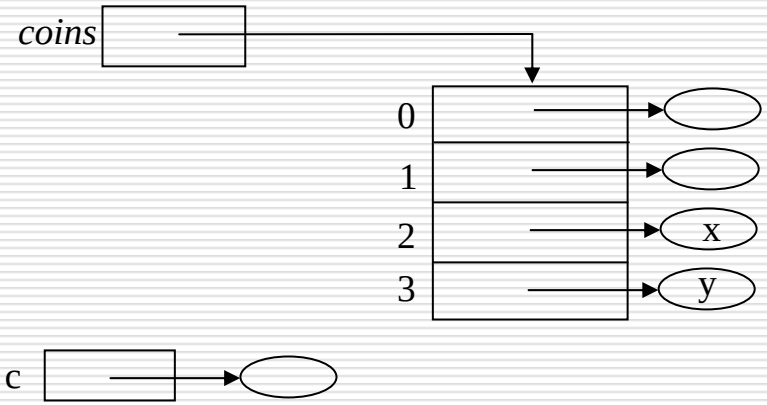
- To set an array list element to a new value, use the *set( )* method of the ArrayList class.

```
Coin aNewCoin = (Coin) new Coin(5.0, "FiveBaht"));
coins.set(1, aNewCoin); // set position 1 of the coins array list to aNewCoin
```

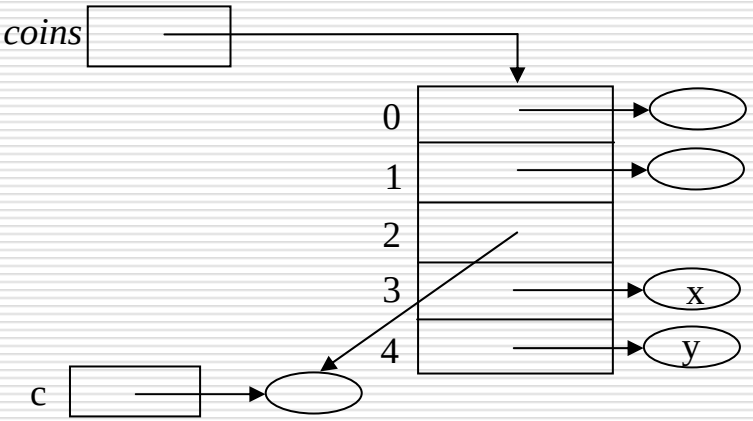


The ArrayList Class

- The *coins.add(i, c)* method adds the object *c* at the position *i* and moves all elements by one position, from the current element at position *i* to the last element in the array list.



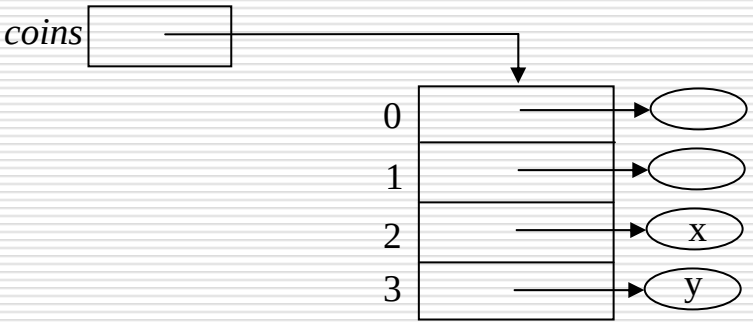
Before calling the *coins.add(2, c)* method, the *coins.size( )* method returns 4.



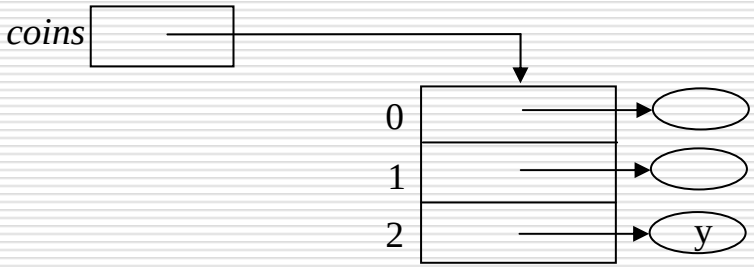
After calling the *coins.add(2, c)* method, the *coins.size( )* method returns 5.

The ArrayList Class

- The *coins.remove(i)* method removes the object at the position *i* and shifts all elements after the removed object by one position.



Before calling the *coins.remove(2)* method, the *coins.size( )* method returns 4.



After calling the *coins.remove(2)* method, the *coins.size( )* method returns 3.

Processing an ArrayList object

- To find an object in an array list, check all elements (objects) until we have found a match.

```
Coin searchedCoin = new Coin(0.5, “FiftySatang”);
boolean match = false;

for (int i = 0; i < coins.size(); i++)
{ Coin aCoin = (Coin) coins.get(i);
   if (aCoin.equals(searchedCoin))
   {   match = true;  break; }
}

if (match)
  System.out.println(“Found the searched coin”);
else
  System.out.println(“Not found the searched coin”);
```

Processing an ArrayList object

- To count the number of certain objects:

```
Coin fiftySatangCoin = new Coin(0.5, “FiftySatang”);
int count = 0;
for (int i = 0; i < coins.size(); i++)
{ Coin aCoin = (Coin) coins.get(i);
  if (aCoin.equals(fiftySatangCoin))
     count++;
}
System.out.println(“There are ” + count + “fifty-Satang coins.”);
```

Processing an ArrayList object

- To find an object with the largest value:

```
if (coins.size() > 0)
{    Coin max = (Coin) coins.get(0);
     for (int i = 1; i < coins.size(); i++)
     {   Coin aCoin = (Coin) coins.get(i);
         if (aCoin.getCoinValue() > max.getCoinValue())
               max = aCoin;
     }
      System.out.println("The largest coin value in the list = " +
                         max.getCoinValue());
}
```

Storing Numbers in Array Lists

- Since numbers and boolean constants are not objects in Java, we cannot create their array lists directly.
- To construct array lists of integers, floating-point numbers or boolean constants, we need to change them into the classes first:

```
double x = 23.45;
Double wrapper = new Double(x);
ArrayList data = new ArrayList();
data.add(wrapper);
```

⇦ change *x* into an object of type Double.

- To retrieve the number, use the *doubleValue( )* method of the *Double* class:

```
Double firstData = (Double) data.get(0);
double x = firstData.doubleValue();
```

- For integers and boolean constants, there are corresponding classes: Integer and Boolean, respectively.