CHAPTER 17

TYPE CASTING

Converting one data type into another data type is called 'type casting' or simply 'casting'. Whenever we assign a value to a variable using assignment operator, the Java compiler checks for uniformity and hence the data types at both the sides should be same. If the data types are not same, then we should convert the types to become same at the both sides. To convert the data type, we use 'cast operator'. Cast operator means writing the data type between simple braces, before a variable or method whose value is to be converted.

Let us revise what is a data type. Data type represents the type of data being stored into a variable (memory). For example, `int x;`

Here, 'int' represents that we can store an integer value into the variable x. So it is called a data type.

Types of Data Types

There are two types of data types, as given here:

- Primitive Data Types or Fundamental Data Types

The data types which represent a single entity (or value) are called 'primitive data types'. For example, take 'int' type, it can store only one integer value. Take 'boolean' value, it can store either true or false - only one value. So they come under primitive data types. The following are the primitive data types:

```
    char, byte, short, int, long, float, double, boolean.
```

- Referenced Data Types or Advanced Data Types

These data types represent several values. For example, take an array. It can store several values. Similarly take a class. It can store different values. So they are called advanced data types. We can access an array or an object of a class in memory through references. So, they are also called 'referenced data types'. The following are examples for referenced data types:

```
    any array, any class (String, StringBuffer, Employee, Manager etc.)
```

Important Interview Question

What is the difference between primitive data types and advanced data types?

Primitive data types represent single values. Advanced data types represent a group of values. Also, methods are not available to handle the primitive data types. In case of advanced data types, methods are available to perform various operations.

We can convert a primitive data type into another primitive data type using casting. Similarly, it is possible to convert a referenced data type into another referenced data type by using casting. Bu we cannot convert a primitive data type into a referenced data type by using casting. For this purpose, methods of Wrapper classes should be used. Wrapper classes will be discussed later in Wrapper Classes chapter.

Casting Primitive Data Types

It is possible to convert one primitive data type into another primitive data type. This is done in tw ways, widening and narrowing. The primitive data types are classified into two types, lower types and higher types. Naturally, the lower types are the types which use less memory and which ca represent lesser number of digits in the value. The higher types use more memory and ca represent more number of digits. To recognize the lower and higher types, the following diagram is useful:

```
    byte,  short,  char, int,  long,  float,  double
    lower <-----------------------------------> higher
```

Thus, char is a lower type than int. float is a higher type than long. `boolean` is not included earli because it cannot be converted into any other type.

Widening in Primitive Data Types

Converting a lower data type into a higher data type is called widening. See the examples:

❑ `char ch = 'A';`

`int num = (int)ch;`   //num contains 65, the ASCII value of 'A'.

Here, we are converting char type variable 'ch' into int type. For this purpose, we used the ca operator by writing 'int' in the simple braces before the variable 'ch'. Now that ch is converted in int type, it can be assigned to the variable num. This is an example for widening.

❑ `int x = 9500;`

`float sal = (float)x;`   //sal contains 9500.0.

Here, we are converting int type variable 'x' into float. So, we wrote (float) before the variable x.

Widening is safe because there will not be any loss of data or precision or accuracy. This is the reason, even though the programmer does not use cast operator, Java compiler does not compla Of course, in this case, Java compiler does the casting operation internally and hence this is als called 'implicit casting'. For example, writing the preceding statements without using cast opera as shown here will compile properly:

❑ `char ch = 'A';`
`int num = ch;`

❑ `int x = 9500;`
`float sal = x;`

Important Interview Question

What is implicit casting?

Automatic casting done by the Java compiler internally is called implicit casting. Implicit casting is done to convert a lower data type into a higher data type.

Narrowing in Primitive Data Types

Converting a higher data type into a lower data type is called 'narrowing'. Take the example:

-
  ```
  int n = 66;
  char ch = (char)n;  //ch contains 'B'.
  ```

Here, we are converting int type n value into char type. The value of n is 66 which is when converted into char type represents the character 'B', since 66 is the ASCII value of 'B'. This character is stored into ch.

-
  ```
  double d = 12.6789;
  int n = (int)d;    //n stores 12.
  ```

Observe, we are converting double type into int type by using the cast operator (int) before the variable d. The value in d is 12.6789. Since it is converted into int type, the fraction part of the number is lost and only 12 is stored in n. Here, we are losing some digits. So narrowing is not safe. This is the reason, Java compiler forces the programmer to use cast operator when going for narrowing. The programmer should compulsory cast the data type. So, narrowing is also called explicit casting.

Important Interview Question

What is explicit casting?

The casting done by the programmer is called explicit casting. Explicit casting is compulsory while converting from a higher data type to a lower data type.

Casting Referenced Data Types

A class is a referenced data type. Converting a class type into another class type is also possible through casting. But the classes should have some relationship between them by the way of inheritance. For example, you can not convert a Dog class into a Horse class, as those classes do not have any relationship between them. But you can convert a College class into a University class, since College is derived from University. And you can convert a Department class into a College, since Department is a sub class of College class. See Figure 17.1 which shows that the classes Department, College and University have relationship by the way of inheritance.

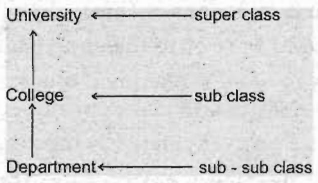

Figure 17.1 Relationship of classes by the way of Inheritance

Generalization and Specialization

Let us take a super class Fruit and the sub classes Citrus and Non-Citrus, as shown in the Figure 17.2.

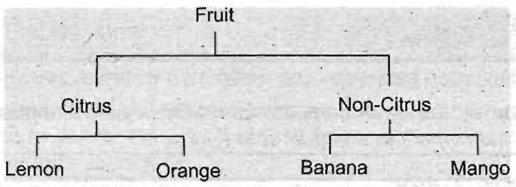

Figure 17.2 The hierarchy of inherited classes from Fruit class

When we talk about a fruit, remember we are talking in general terms and it may represent
kind of fruit. So, here the scope is widened. Suppose, we talk about citrus fruit, then we came d
one step in inheritance hierarchy and thus, we have eliminated other types of fruit. So, we
becoming more specific. When we still come down to lemon, we are pin-pointing the type of
fruit. It is lemon only and not any other fruit. This is very specific. This means when we come d
from super class to sub classes, we are becoming more and more specific. When we go back f
sub classes to super class, we are becoming more general.

Converting a sub class type into a super class type is called 'generalization', because we are ma
the sub class to become more general and its scope is widening. This is also called widening or
casting. Widening is safe because the classes will become more general. For example, if we
lemon is a fruit, there will be no objection. Hence, Java compiler will not ask for cast operat
generalization. It will do implicit casting. Similarly, converting a super class type into a sub c
type is called 'specialization'. Here, we are coming down from more general form to a specific
and hence the scope is narrowed. Hence, this is also called narrowing or down-casting. Narrowi
not safe because, the classes will become more and more specific thus giving rise to more and
doubts. For example, if we say so and so fruit is a citrus fruit, we should show proof fo
statement, since we are becoming more specific. In this case, Java compiler specifically asks
programmer to use cast operator.

Important Interview Question

What is generalization and specialization?

Generalization is a phenomenon where a sub class is promoted to a super class, and hen becomes more general. Generalization needs widening or up-casting. Specialization is phenomen where a super class is narrowed down to a sub class. Specialization needs narrowing or down-castin

Generally, any class reference can be used to refer to that class object only. For example,

- ❑ `Employee e;` //e is the reference variable of Employee class.
- ❑ `e= new Employee();` //here, e is used to refer to Employee class object.

Suppose class One and class Two are two related classes:

- ❑ class One //One is super class.
- ❑ class Two extends One //Two is sub class of One.

Now, the super class reference is used to refer to the super class object.

- ❑ `One o = new One(); //class One's` reference o is referring to One's object. Similarl class reference can be used to refer to sub class object.
- ❑ `Two t = new Two();` //class Two's reference t is used to refer to Two's object

In the preceding two cases, we do not require any casting. The reason is that at the left side
the right side of the assignment operator, we have same data types. For example,

```
    One o = new One();
```

Here, 'o' is the reference of class One. So the data type of 'o' is One. At the right side of assignment, we got class One's object. So the data type (class) of the object is also One. So casting is not needed here.

But when, we try to use a reference to refer to a different class object, we need casting. For example, if class One's reference is used to refer to class Two's object, as:

```
One o = new Two();
```

At the left side, we got the reference 'o' whose data type is class One. At the right side, we got the object whose data type is class Two. Now it is essential, we convert the object's data type into class One using cast operator, as:

```
One o = (One) new Two();  //convert class Two's type as class One
```

Here, sub class object type is converted into super class. This is an example for widening or up-casting. In this case, even if we do not use cast operator, there will not be any error message, as the Java compiler will do implicit casting. Similarly, we can convert class One's type as class Two's type in this case:

```
Two t = (Two) new One();  //convert class One's type as class Two
```

Here, we are converting from super class to sub class, which comes under narrowing or down-casting. In this case, cast operator is compulsory as this is unsafe operation.

dening in Referenced Data Types

Let us observe the difference between widening and narrowing in case of referenced data types. In the program 1, we take class One as super class and class Two is its sub class. We do widening by using super class reference to refer to sub class object. In this case, we convert the sub class object type as super class type.

Program 1: Write a program to see the widening effect where super class reference is used to refer to sub class object.

```
//Widening using referenced data types
class One
{
	void show1()
	{
		System.out.println("Super class method");
	}
}
class Two extends One
{
	void show2()
	{
		System.out.println("Sub class method");
	}
}
class Cast
{
	public static void main(String args[ ])
	{
		One o;  //o is super class reference
		o = (One)new Two(); //o is referring to sub class object
		//the above is widening
		o.show1();
	}
}
```

Output:

```
C:\> javac Cast.java
C:\> java Cast
Super class method
```

In program 1, we used super class reference to refer to sub class object. In this case, the sub class object type is converted into super class type, as:

```
o = (One) new Two();
```

Please observe that, we are able to call the `show1()` method of the super class. But in this case, it is not possible to call the `show2()` method of the sub class. For example, if we call the sub class `show2()` method as:

```
o.show2();
```

There would be an error message during compilation time, as:

```
Cast.java:26: cannot find symbol
symbol: method show2()
location: class One
        o.show2();
1 error
```

So, in widening, the programmer can access all the super class methods, but not the sub class methods. Suppose, we override the super class methods in sub class, as shown here in Program 2 then it is possible to access the sub class methods but not the super class methods. Any how, the programmer will get only 50% functionality into his hands.

Program 2: Let us make a program to override the super class `show1()` method in sub class. Now only the sub class method is executed.

```
//Widening in referenced data types
class One
{
      void show1()
      {
              System.out.println("Super class method");
      }
}
class Two extends One
{
      void show1()  //override the super class method
      {
              System.out.println("Sub class method");
      }
}

class Cast
{
      public static void main(String args[ ])
      {
              One o;  //o is super class reference
              o = (One)new Two(); //o is referring to sub class object
              //the above is widening
              o.show1();

      }
}
```

Output:

```
C:\> javac Cast.java
C:\> java Cast
Sub class method
```

Narrowing in Referenced Data Types

Narrowing represents converting super class type into sub class type. In the following program, we attempt to go for narrowing by taking sub class reference to refer to super class object. See the Program 3.

Program 3: Write a program for creating sub class reference which is used to refer to the super class object.

```
//Narrowing using super class object
class One
{
        void show1()
        {
                System.out.println("Super class method");
        }
}
class Two extends One
{
        void show2()
        {
                System.out.println("Sub class method");
        }
}

class Cast
{
        public static void main(String args[ ])
        {
                Two t;  //t is sub class reference
                t = (Two)new One(); //t is referring to super class object
                //the above is narrowing
                t.show1();

        }
}
```

Output:

```
C:\> javac Cast.java
C:\> java Cast
Exception in thread "main" java.lang.ClassCastException:
One cannot be cast to Two
```

By observing the output of Program 3, we can understand that the method call `t.show1()` is not executing the super class method. Similarly, write another statement and call the sub class method `show2()` as:

```
t.show2()
```

which will also give the same error. So, in narrowing using super class object, we cannot access any of the methods of the super class or sub class. So the programmer will get 0% functionality in this case.

There is a solution for this problem. Let us not create an object to super class, as we did in the previous case. This time, we create an object to sub class and use narrowing, as shown in the Program 4.

Program 4: Write a program for creating super class reference to refer to sub class object.

```
//Narrowing using sub class object
class One
{
        void show1()
        {
                System.out.println("Super class method");
        }
}
class Two extends One
{
        void show2()
        {
                System.out.println("Sub class method");
        }
}

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

                One o;
                o = new Two(); //super class reference to refer to sub class object
                Two t = (Two)o; //this is narrowing - convert class One's reference
                //type as class Two's type
                t.show1();
                t.show2();
        }
}
```

Output:

```
C:\> javac Cast.java
C:\> java Cast
Super class method
Sub class method
```

In program 4, we convert the super class reference type into sub class reference type. Then using the sub class reference, we try to call the methods. From the preceding code, it is evident that if an object to sub class is created, it is possible to access all the methods of the super class as well as the sub class. Narrowing using sub class object will provide 100% functionality to be accessible to the programmer. This is the reason in inheritance, we create an object always to sub class, but not to the super class.

Summary

So the following points can be summarized from the preceding discussion:

- If the super class reference is used to refer to super class object, naturally all the methods of super class are accessible.
- If the sub class reference is used to refer to sub class object, all the methods of the super class as well as sub class are accessible since the sub class object avails a copy of the super class.
- If widening is done by using sub class object, only the super class methods are accessible. If they are overridden, then only the sub class methods are accessible.

- If narrowing is done by using super class object, then none of the super class or sub class methods are accessible. This is useless.
- If narrowing is done by using sub class object, then all the methods of both the super and sub classes are available to the programmer.

From the preceding points, we can understand that since an object behavior is changed depending on which reference is used to refer the object, we can say that the object is exhibiting polymorphism.

Important Interview Question

What is widening and narrowing?

Converting lower data type into a higher data type is called widening and converting a higher data type into a lower type is called narrowing. Widening is safe and hence even if the programmer does not use cast operator, the Java compiler does not flag any error. Narrowing is unsafe and hence the programmer should explicitly use cast operator in narrowing.

The Object Class

There is a class with the name 'Object' in `java.lang` package which is the super class of all classes in Java. Every class in Java is a direct or indirect sub class of the Object class. The Object class defines the methods to compare objects, to convert an object into a string, to notify threads (processes) regarding the availability of an object, etc.

Important Interview Question

Which is the super class for all the classes including your classes also?

Object class

Object class reference can store any reference of any object. This becomes an advantage when we want to write a method that needs to handle objects of unknown type. If we define a parameter of Object type, any class object can be passed to the method. Thus, the method can receive any type of object and handle it. The methods of Object class are given in the Table 17.1.

Table 17.1

Method	Description
`equals()`	This method compares the references of two objects and if they are equal, it returns true, otherwise false. The way it compares the objects is dependent on the objects.
`toString()`	This method returns a string representation of an object. For example, the string representation of an Integer object is an integer number displayed as string.
`getClass()`	This method gives an object that contains the name of a class to which an object belongs.
`hashCode()`	This method returns hash code number of an object.
`notify()`	This method sends a notification to a thread which is waiting for an object.
`notifyAll()`	This method sends a notification for all waiting threads for the object.

Method	Description
`wait()`	This method causes a thread to wait till a notification is received from a `notify()` or `notifyAll()` methods.
`clone()`	This method creates a bit wise exact copy of an existing object.
`finalize()`	This method is called by the garbage collector when an object is removed from memory.

Let us see how to compare two objects by using `equals()` method of Object class. This meth normally compares the references of two objects. If both the references refer to same object, then gives true, otherwise it gives false. But in case of String objects and wrapper class objec (Character, Integer, Float, Double etc. classes are called wrapper classes), it compares the conten of the objects. If the contents are same then it returns true, otherwise false. In Program 5, we a using `equals()` method to compare two objects of a general class 'Myclass' and two objects wrapper class 'Integer'. In case of `Myclass` objects, *if the* references are same, then `equals` method gives true, otherwise false. In case of Integer class objects, it gives true if the contents a same, otherwise false.

Program 5: Write a program where `equals()` method compares `Myclass` objects' references. T same `equals()` method is used to compare Integer class objects' contents.

```
//equals() method
//take Myclass that stores an int value
class Myclass
{
        int x;

        Myclass(int x)
        {
                this.x = x;
        }
}

class Compare
{
        public static void main(String args[ ])
        {
                //create two Myclass objects with same content.
                In this case, references of
                //objects will be different.
                Myclass obj1 = new Myclass(15);
                Myclass obj2 = new Myclass(15);

                //create two wrapper class objects and store same content.
                In this case,
                //references of objects will be different.
                Integer obj3 = new Integer(15);
                Integer obj4 = new Integer(15);


                if(obj1.equals(obj2))
                System.out.println("obj1 and obj2 are same");
                else System.out.println("obj1 and obj2 are not same");

                if(obj3.equals(obj4))
                System.out.println("obj3 and obj4 are same");
                else System.out.println("obj3 and obj4 are not same");

        }
}
```

Output:

```
C:\> javac Compare.java
C:\> java Compare
obj1 and obj2 are not same
obj1 and obj4 are same
```

Let us use getClass() method to know the name of the class to which an object belongs. In Program 6, we are taking Myclass which contains an int value. There is another class KnowName which contains a static method printName() as:

```
static void printName(Object obj)
{
	Class c = obj.getClass();
	String name = c.getName();
	System.out.println("The classname= "+ name);
}
```

Observe that this method has a parameter which is the reference variable of Object class. Object class reference can be used to refer to any other class object. So it is possible to send Myclass object to this method, and then the Object class reference will refer to Myclass object.

❑ Observe the code in printName() method:

```
Class c = obj.getClass();
```

Here, getClass() method gets the class name of the object and it stores that class name in the object of the class Class. Please note that there is a class with the name 'Class' in Java. This class exists in java.lang package. Now using c.getName(), we can find the class name.

Program 6: Write a program where an object is passed to printName() method and the class name of the object is displayed by the method.

```
//Using getClass() to know the classname
//Myclass stores an int value
class Myclass
{
	int x;

	Myclass(int x)
	{
		this.x = x;
	}
}
//This class contains method to receive an object and display the classname
class KnowName
{
	static void printName(Object obj)
	{
		//get the class name into an object c of the class Class
		Class c = obj.getClass();
		//get the name of the class using getName()
		String name = c.getName();
		System.out.println("The classname= "+ name);
	}
}

class Demo
{
	public static void main(String args[ ])
	{
		//create Myclass object obj
		Myclass obj = new Myclass(10);
```

```
                    //know the class name of the object obj by calling printName().
                    KnowName.printName(obj);
            }
    }
```

Output:

```
    C:\> javac Demo.java
    C:\> java Demo
    The classname= Myclass
```

Cloning the Class Objects

The process of creating an exact copy of an existing *object is called 'cloning'. In cloning,* already a object should exist and when we clone the object, a bit wise copy of the object will result. The original object and the cloned object will be exactly the same bit to bit. If the original object has some data in it, it also automatically comes into cloned object.

There are two types of cloning. When the cloned object is modified, same modification will also affect the original object. This is called 'shallow cloning'. When the cloned object is modified, if the original object is not modified, then it is called 'deep cloning'.

When we have new operator to create the objects, why do we need the cloning technology? Let us take an example of object1 which is created by using new operator. There is a lot of processing done on this object, so the content of the object has been drastically changed. Let us call it object2. At this intermediate stage, we want another copy of this object. There are two ways to perform this:

- Using new operator, we can create another object. But, when new operator is used to create the object, the object is created by using the initial values as object1. So, the same processing should be repeated on this object to get the intermediate object, i.e., object2.
- The other way is to clone the object2, so that we get exact copy of the object. This preserves a copy of the intermediate object and hence the original object and the cloned objects can be processed separately. This method is easy because, we can avoid a lot of processing to be done on the object.

From the preceding discussion, we can understand that cloning is advantageous than using new operator while creating objects.

Let us discuss the steps to clone the object of a class:

- The class whose objects to be cloned should implement Cloneable interface. This interface belongs to `java.lang` package. Cloneable interface indicates that the class objects are cloneable. If this interface is not implemented, then that class objects can not be cloned and it gives `CloneNotSupportedException`.
- `clone()` method of Object class is used for cloning. Object is the super class for every class. So, let us write our own method in the class and call the `clone()` method, something like this:

```
    public Object myClone()  //our own method
    {
        return super.clone();  //create cloned object and return it
    }
```

- It is also possible to create a cloned object, by overriding the `clone()` method of Object class. Since `clone()` method is defined as protected Object `clone()` in Object class, we can write our `clone()` method overriding this, as:

```
    protected Object clone()  //This method overrides clone() method of Object class
    {
        return super.clone();
    }
```

Here, we are overriding the `clone()` method and again we are calling the `clone()` method from it. These concepts are used in Program 7 to clone Employee object. In this program, we are creating Employee class with id and name details. The Employee class should implement Cloneable interface because we are going to clone Employee class objects. There is `myClone()` method which calls the super class (Object class) `clone()` method to clone the Employee class object.

Program 7: Write a program to make cloning Employee class object by writing our own `myClone()` method, from where Object class `clone()` method is called.

```
    //Cloning example
    class Employee implements Cloneable
    {
            //instance vars
            int id;
            String name;

            //constructor to initialize vars
            Employee(int id, String name)
            {
                    this.id = id;
                    this.name = name;
            }

            //method to display the details
            void getData()
            {
                    System.out.println("Id= "+ id);
                    System.out.println("Name= "+ name);
            }

            //clone the present class object
            public Object myClone() throws CloneNotSupportedException
            {
                    return super.clone();
            }
    }


    class CloneDemo
    {
            public static void main(String args[ ])  throws CloneNotSupportedException
            {
                    //create Employee class object using new operator
                    Employee e1 = new Employee(10, "Srinivas");

                    System.out.println("Original object:");
                    e1.getData();

                    //create another object by cloning e1. As myClone() method returns
                    //object of Object class type, it should be converted into
                    Employee type
                    Employee e2 = (Employee)e1.myClone();

                    System.out.println("Cloned object: ");
                    e2.getData();
            }
    }
```

Output:

```
    C:\> javac CloneDemo.java
    C:\> java CloneDemo
    Original object:
```

```
Id= 10
Name= Srinivas
Cloned object:
Id= 10
Name= Srinivas
```

We can also rewrite the preceding program, by overriding the `clone()` method of Object . Replac the `myClone()` method code with the following code:

```
protected Object clone() throws CloneNotSupportedException
 {
    return super.clone();
 }

And call this method from main() as:
Employee e2 = (Employee)e1.clone();
```

Important Interview Question

Which method is used in cloning?

`clone()` *method of Object class is used in cloning.*

In cloning the objects, the class should implement Cloneable interface. In fact, the Cloneabl interface does not have any members. It means, this interface does not have any variables o methods. Such an interface is called 'marking interface' or 'tagging interface'. This interface behave like a 'tag' indicating the class objects are cloneable. Or, we can say that this interface is 'marking the class so that any body can clone the class objects. If we do not implement Cloneable interfac then that class objects are not cloneable and `clone()` method give `CloneNotSupportedException`.

Important Interview Question

Can you write an interface without any methods?

Yes.

What do you call the interface without any members?

An interface without any members is called marking interface or tagging interface. It marks the class objects for a special purpose. For example, Cloneable (`java.lang`) and Serializable (`java.io`) are two marking interfaces. Cloneable interface indicates that a particular class objects are cloneable while Serializable interface indicates that a particular class objects are serializable.

We shall discuss the `wait()`, `notify()` and `notifyAll()` methods of Object class later in a chapter on Threads.

Conclusion

Generally, a class reference can be used to refer to any of that class objects. By casting th reference, we can use it to refer to its sub or super class objects. In this case, the behavior of th object will change exhibiting the polymorphic nature. We can cast primitive data types in two ways widening and narrowing. Similarly, the advanced data types also can be casted in two ways widening and narrowing. Finally, the universal super class 'Object' comes into picture as all othe classes in Java have been derived from Object class. So, Object class reference can be used to refe to any other class.