Chapter 16

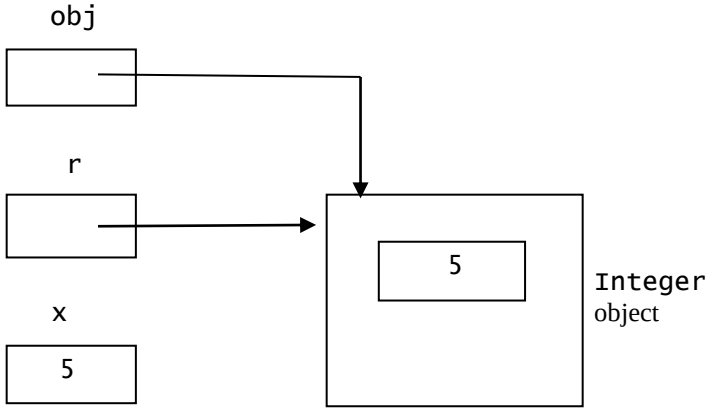
Generic Programming Part 1

What is generic programming?

Programming structures that can accommodate a variety of data types.

```
Object obj;
int x = 5;
Integer r = new Integer(x);
obj = r;
```

`Obj` is a generic pointer

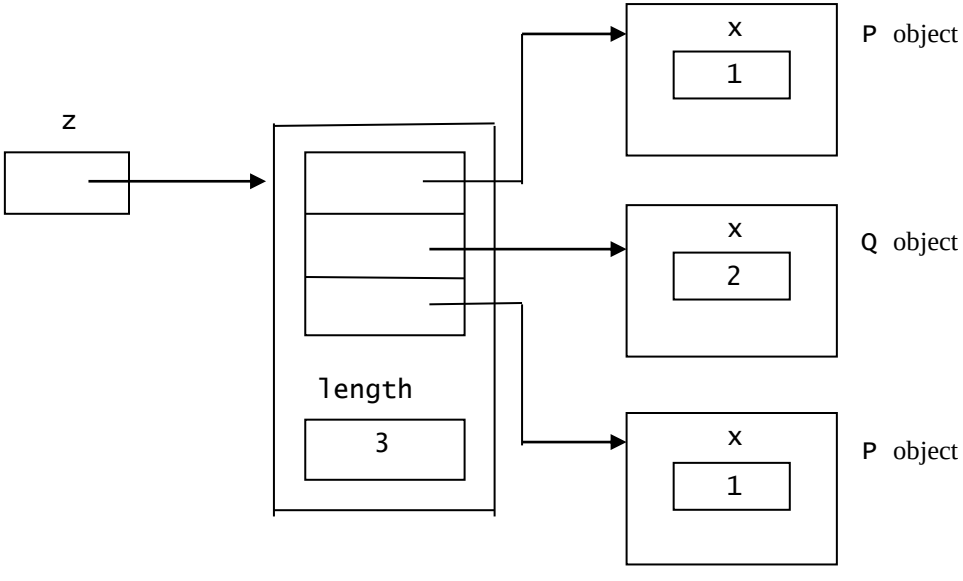

obj
r
x
5
5
Integer
object

Using an `Object` array

```
 1 class P
 2 {
 3    private int x = 1;
 4    public void xDisplay()
 5    {
 6       System.out.println("x = " + x);
 7    }
 8 }
 9 //============================================
10 class Q
11 {
12    private int x = 2;
13    public void xDisplay()
14    {
15       System.out.println("x = " + x);
16    }
17 }
```

```
19 class Generic1
20 {
21    public static void main(String[] args)
22    {
23       Object[] z = new Object[3];
24
25       z[0] = new P();
26       z[1] = new Q();
27       z[2] = new P();
28
29       for (int i = 0; i < z.length; i++)
30       {
31          if (z[i] instanceof P)
32             ((P)z[i]).xDisplay();
33          else
34          if (z[i] instanceof Q)
35                ((Q)z[i]).xDisplay();
36       }
37    }
38 }
```

```
25        z[0] = new P();        // init array with objects
26        z[1] = new Q();        // of different types
27        z[2] = new P();
```



Casts needed

```
29        for (int i = 0; i < z.length; i++)
30        {
31           if (z[i] instanceof P)
32              ((P)z[i]).xDisplay();
33           else
34           if (z[i] instanceof Q)
35                 ((Q)z[i]).xDisplay();
36        }
```

```
 1 class R
 2 {
 3    public void xDisplay()
 4    {
 5       // dummy method
 6    }
 7 }
 8 //================================================
 9 class P extends R             // P subclass of R
10 {
11    private int x = 1;
12    public void xDisplay()
13    {
14       System.out.println("x = " + x);
15    }
16 }
17 //================================================
18 class Q extends R             // Q subclass of R
19 {
20    private int x = 2;
21    public void xDisplay()
22    {
23       System.out.println("x = " + x);
24    }
25 }
```

```
27 class Generic2
28 {
29    public static void main(String[] args)
30    {
31       R[] z = new R[3];
32
33       z[0] = new P();
34       z[1] = new Q();
35       z[2] = new P();
36
37       for (int i = 0; i < z.length; i++)
38            z[i].xDisplay();
39    }
40 }
```

Abstract classes

- Cannot be instantiated.
- Subclasses of an abstract class can be instantiated if all abstract methods in the abstract class are overrided with concrete methods.

```
 1 abstract class Picasso
 2 {
 3    abstract public void xDisplay();
 4 }
 5 //================================================
 6 class P extends Picasso
 7 {
 8    private int x = 1;
 9    public void xDisplay()
10    {
11       System.out.println("x = " + x);
12    }
13 }
14 //================================================
15 class Q extends Picasso
16 {
17    private int x = 2;
18    public void xDisplay()
19    {
20       System.out.println("x = " + x);
21    }
22 }
```

```
24 class Generic3
25 {
26    public static void main(String[] args)
27    {
28       Picasso[] z = new Picasso[3];
29
30       z[0] = new P();
31       z[1] = new Q();
32       z[2] = new P();
33
34       for (int i = 0; i < z.length; i++)
35            z[i].xDisplay();
36    }
37 }
```

Interfaces

An abstract class can contain

- abstract methods
- non-abstract methods
- constructors
- instance variables
- static variables
- named constants, both static and non-static

Interface can contain

- abstract methods
- named constants that are public, static, and final.

```
abstract interface Sample
{
   public static final double PI = 3.14159;
   abstract public void f();
}
```

equivalent to

```
interface Sample
{
   double PI = 3.14159;
   public void f();
}
```

```
 1 interface I
 2 {
 3    public void xDisplay();
 4 }
 5 //================================================
 6 class P implements I
 7 {
 8    private int x = 1;
 9    public void xDisplay() // overrides abst method
10    {
11       System.out.println("x = " + x);
12    }
13 }
14 //================================================
15 class Q implements I
16 {
17    private int x = 2;
18    public void xDisplay()
19    {
20       System.out.println("x = " + x);
21    }
22 }
```

```
24 class Generic4
25 {
26    public static void main(String[] args)
27    {
28       I[] z = new I[3];
29
30       z[0] = new P();
31       z[1] = new Q();
32       z[2] = new P();
33
34       for (int i = 0; i < z.length; i++)
35            z[i].xDisplay();
36    }
37 }
```

Using an interface to provide constants to a class

```
interface MyConstants
{
   double RATE = 7.12;
   double MINBALANCE = 100.0;
}
//=========================================
class U implements MyConstants
{
    // can use RATE and MINBALANCE here
}
//=========================================
class V implements MyConstants
{
   // can use RATE and MINBALANCE here
}
```

Generic Programming Part 2

Generic classes

```
 1 class OneInteger
 2 {
 3    private Integer x; // type of x is always Integer
 4    //---------------------------------
 5    public OneInteger(Integer xx)
 6    {
 7       x = xx;
 8    }
 9    //---------------------------------
10    public Integer get()
11    {
12       return x;
13    }
14 }
15 //================================================
16 class TestOneInteger
17 {
18    public static void main(String[] args)
19    {
20       OneInteger p = new OneInteger(7);
21       System.out.println(p.get());
22    }
23 }
```

```
 1 class OneThing<T>  // T is the type parameter
 2 {
 3    private T x;    // Use T as if it is a type
 4    //---------------------------------
 5    public OneThing(T xx)
 6    {
 7       x = xx;
 8    }
 9    //---------------------------------
10    public T get()
11    {
12       return x;
13    }
14 }
15 //================================================
16 class TestOneThing
17 {
18    public static void main(String[] args)
19    {
20       OneThing<Integer> p1;
21       p1 = new OneThing<Integer>(7);
22       System.out.println(p1.get());
23
24       OneThing<String> p2 = new OneThing<String>("hello");
25      System.out.println(p2.get());
26   }
27 }
```

What you can and cannot do in a generic class

```
OneThing<int> p1;                 // Illegal
p1 = new OneThing<int>();         // Illegal
```

```
 1 class Hasf
 2 {
 3    public void f()
 4    {
 5       System.out.println("hello");
 6    }
 7 }
 8 //================================================
 9 class BadGeneric1<T>
10 {
11    T r;
12    //-----------------------------------
13    public BadGeneric1(T rr)
14    {
15       r = rr;
16    }
17    //-----------------------------------
18    public void m()
19    {
20       System.out.println(r.toString());  // legal
21       r.f();                             // illegal
22    }
23 }
```

```
25 class TestBadGeneric1
26 {
27    public static void main(String[] args)
28    {
29       Hasf h1 = new Hasf();
30       BadGeneric1<Hasf> g1 = new BadGeneric1<Hasf>(h1);
31       g1.m();
32       BadGeneric1<String> g2 = new BadGeneric1<String>("hello");
33       g2.m();
34    }
35 }
```

```
 1  class BadGeneric2<T>
 2  {
 3     T t  = new T();                    // illegal
 4     T[] a = (T[]) new T[20];           // illegal
 5  }
 6  //================================================
 7  class GoodGeneric1<T>
 8  {
 9     T t  = (T)new Object();            // legal
10     T[] a = (T[]) new Object[20];      // legal
11  }
12  //================================================
13  class GoodGeneric2<T extends C>       // C is a class
14  {
15     T t  = (T)new C();                 // legal
16     T[] a = (T[]) new C[20];           // legal
17  }
```

Extending the type parameter with the `Comparable` interface

```
 1 class OrderedPair<T>
 2 {
 3    private T x, y;
 4    //---------------------------------
 5    public OrderedPair(T xx, T yy)
 6    {
 7       x = xx;
 8       y = yy;
 9    }
10    //---------------------------------
11    public T smaller()
12    {
13       if (x.compareTo(y) < 0)     // illegal
14          return x;
15       else
16          return y;
17    }
18 }
19 //================================================
20 class TestOrderedPair
21 {
22    public static void main(String[] args)
23    {
24       OrderedPair<Integer> p = new OrderedPair<Integer>(1, 2);
25       System.out.println(p.smaller());
26    }
27 }
```

How to make line 13 legal

Replace

```
1 class OrderedPair<T>
```

with

```
class OrderedPair<T extends Comparable>
```

Writing our own `ArrayList` class

```
 1 class MyArrayList<T>
 2 {
 3    private T[] current = (T[])new Object[10];
 4    private int size = 0;
 5    //---------------------------------
 6    public T get(int i)
 7    {
 8       return current[i];
 9    }
10    //---------------------------------
11    public void add(T x)
12    {
13       if (size >= current.length)
14       {
15          T[] old = current;   // save current
16          current = (T[]) new Object[old.length + 50]; // new array
17          for (int i = 0; i < old.length; i++)   // copy
18             current[i] = old[i];
19       }
20       current[size++] = x;   // add x to array
21    }
22    //---------------------------------
23    public int size()
24    {
25       return size;
26    }
27 }
```

```
29 class TestMyArrayList
30 {
31    public static void main(String[] args)
32    {
33       MyArrayList<Integer> oal = new MyArrayList<Integer>();
34       oal.add(5);
35       oal.add(2);
36       oal.add(3);
37       System.out.println(oal.get(0));
38       System.out.println(oal.get(1));
39       System.out.println(oal.get(2));
40    }
41 }
```

Iterators

```
 1 import java.util.*;    // for ArrayList, Iterator
 2 class IteratorExample
 3 {
 4    public static void main(String[] args)
 5    {
 6       ArrayList<String> sal;
 7       sal = new ArrayList<String>();
 8
 9       sal.add("Bert");
10       sal.add("Ernie");
11       sal.add("Grover");
12
13       Iterator<String> itr;
14       itr = sal.iterator();     // get iterator
15
16       while (itr.hasNext())
17          System.out.println(itr.next());
18    }
19 }
```