Chapter 7:The Java Object Model

Chapter Topics

- The Java Type System
- Type Inquiry
- The Object Class
- Shallow and Deep Copy
- Serialization
- Reflection
- Generic types

7.1 The Java Type Systems

1. Types and Subtypes

Types

- A type specifies set of values and the operations that can be applied to the values.
- Strongly typed language
  1. Every variable has a type
  2. All assignments are checked for type compatibility

- Every type in Java is one of the following:
  - Primitive types: byte, short, int, long, char, float, double, boolean
  - Class types: Rectangle, BankAccount
  - Interface types: Shape, Comparable
  - Array types: double[][]
  - Note: void is not a type
- Every variable has a type and the type indicates what values the variable can hold.
  - Primitive type variable: primitive value

    e.g.) 13, 10.2, 10.2F, ‘a’, false

  - Class type variable: reference to an object of a class or null

    e.g.) new Rectangle(5, 10, 20, 30), "Hello"

  - Interface type: reference to an object of a class that implements the interface

    e.g.) Comparable c = new String(“A”);

  - Array type variable: reference to an array or null

e.g.) int[] data = {2, 3, 5, 7, 11, 13 }

Subtypes

- S is a subtype of T if
  - S and T are the same type

    e.g.) BankAccount b = new BankAccount();
  - S and T are both class types, and T is a direct or indirect superclass of S

    e.g.) Person p = new Employee(10);
  - S is a class type, T is an interface type, and S or one of its superclasses implements T

    e.g.) Comparable c = new String("A");
  - S and T are both interface types, and T is a direct or indirect super interface of S

    e.g.) public interface TypeT{ … };
    public interface TypeS extends TypeT { … };
    public class ConcreteC implements TypeS { … };

    TypeT t = new ConcreteC();
  - S and T are both array types, and the component type of S is a subtype of the component type of T

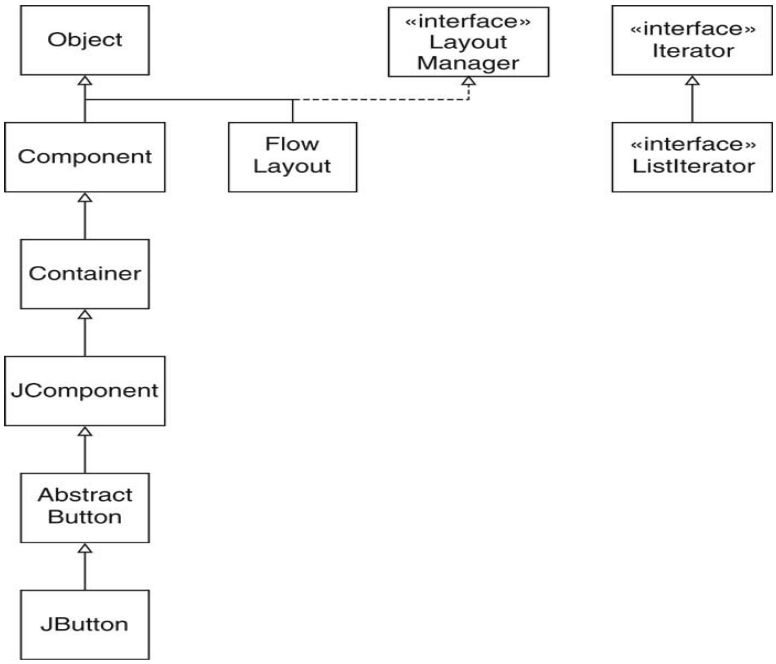
    e.g.) public Person max(Person[] p)
    { Person max = p[0];
    for (int i = 1; i < p.length; i++)
    if (p[i].getAge() > max.getAge()) max = p[i];
    }

    [Q] Is the following code segment valid ?
    Employee[] e = new Employee[2];
    e[0] = new Employee(36, …);
    e[1] = new Employee(50, …);
    max(e); // Person[] p = e;

- S is not a primitive type and T is the type Object
- S is an array type and T is Cloneable or Serializable

- Subtype Examples
  - Container is a subtype of Component
  - JButton is a subtype of Component
  - FlowLayout is a subtype of LayoutManager
  - ListIterator is a subtype of Iterator
  - Rectangle[] is a subtype of Shape[]
  - int[] is a subtype of Object
  - int is *not* a subtype of long
  - long is *not* a subtype of int
  - int[] is *not* a subtype of Object[]
  - int[] is not a subtype of double[]

  double[] d = new int[10]; // incompatible type error message



2. Array Types

- S[] is a subtype of T[] when S is a subtype of T.

For example,

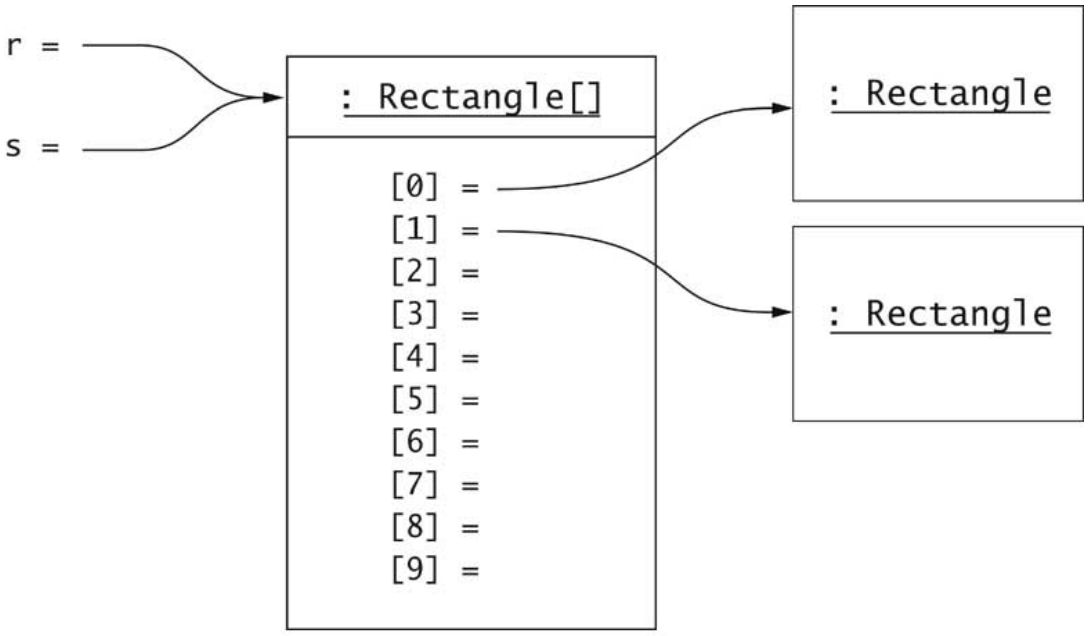
```
Rectangle[] r = new Rectangle[10];
Shape[] s = r; // Both r and s are references to the same
array
```

- Each array object remembers its component type.

For example, when you create an array using new Rectangle[10], the array element should hold a rectangle. (That is an object that *is* a rectangle). The assignment s[0] = new Polygon(); compiles but throws an ArrayStoreException at runtime.

```
import java.awt.*;
import javax.swing.text.*;
public class Test
{  public static void main (String [] args)
   {
      Shape[] s = new Rectangle[2];
      s[0] = new Polygon(); // error
      s[1] = new DefaultCaret(); // DefaultCaret is Rectangle

   }
}
```



3. Wrapper Classes

- Primitive types aren't classes. In other words, you cannot think of "is-a relationship" between primitive types and classes.
- If objects are expected, use wrapper classes. There are eight wrapper classes, one for each primitive type.

```
Byte Short Integer Long Character Float Double Boolean
```

- Java 5 features: Auto-boxing and auto-unboxing

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(13); // calls new Integer(13)
int n = numbers.get(0); // calls intValue();
```

4. Enums

4.1 Before Java 5.0

Before Java 5.0, an enumerated type was represented by a set of integer constants.

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

This pattern has problems

- **Not typesafe**

int season = -9;
int season = SEASON_WINTER + SEASON_SPRING ;

- **Printed values are uninformative** - Because they are just ints, if you print one out all you get is a number, which tells you nothing about what it represents, or even what type it is.

4.2 Java 5.0 enum

In addition to solving the problems mentioned, the enum type allows you to add arbitrary methods and fields to an enum type, to implement arbitrary interfaces, and more.

(1) Type for which all the values for the type are known when the type is defined.
(2) Creating an enumerated type involves three basic components

1) The `enum` keyword
2) A name for the new type
3) A list of allowed values for the type

Example:

```
public enum Grade
{ A, B, C, D, F, INCOMPLETE }
```

Once an enum type is created, it can be used to declare a variable as shown below.

```
public class Student
```

```
{
  private String firstName;
  private String lastName;
  private Grade grade;


  public Student(String firstN, String lastN)
  {
    firstName = firstN;
    lastName = lastN;
  }

  public void assignGrade(Grade g)
  {  grade = g; }

  public Grade getGrade()
  {  return grade; }
}
```

(3) Features of Java Enum types

- Enums are classes – you get type-safety, compile-time checking, and the ability to use them in variable declarations. In fact, the compiler sort of translates the enum type into a class. (See the example below.)
- Enums extend java.lang.Enum
- Enums implements java.lang.Comparable
- Enums have no public constructor
- Enum values are public, static, and final
- Enums inherit toString() from java.lang.Enum and can override it to return a more user friendly name of the value.
- Enum privide a `valueOf()` method that returns the enum constant with the given name
- Enums provide a `values()` method that returns an array containing each of the enum constants in the order where they were declared.

```
public enum Suit
{ HEART,  DIAMOND, CLUB, SPADE }
```

Then, the compiler produces the following class.

```
public class Suit extends Enum<Suit>
{
    public static final Suit HEART =
                    new Suit (“HEART”, 0);
    public static final Suit DIAMOND =
                    new Suit(“DIAMOND”, 1);
    public static final Suit CLUB =
                    new Suit(“CLUB”, 2);
    public static final Suit SPADE =
```

```
                    new Suit("SPADE", 3);

    private Suit(String name, int ordinal) { super(name,
  ordinal); }

    public static Suit valueOf(String name) {…}
    public static Suit[] values() { … }

  }
```

- An enum can have all the class members: fields, methods, constructors, and nested types. A constructor of an enum class must be private.

```
public enum Size
{    SMALL(10), MEDIUM(20), LARGE(30) ;

     private double value;

     private Size(double value)
     { this.value = value; }

     public double getValue() { return value; }


}
```

Then, the compiler transforms it into something like this Java class. The compiler will
- Add two static utility methods valueOf and values.
- Add "name" and "ordinal" arguments to every constructor declaration and constructor call.

```
public class Size extends Enum <Size>
{
   public static final Size SMALL = new Size("SMALL", 0, 10);
   public static final Size MEDIUM = new Size("MEDIUM",1, 20);
   public static final Size LARGE = new Size("LARGE", 2, 30);

   private double value;

   private Size(String name, int ordinal, double value)
   { super(name, ordinal);
     this.value = value;
   }

   public double getValue() { return value; }

    public static Size[] values() { … }
    public static Size valueOf(String name) {…}
}
```

- Example: Use of an enum type

```
public enum Planet
{
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius)
    {
        this.mass = mass;
        this.radius = radius;
    }
    public double mass()   { return mass; }
    public double radius() { return radius; }

    // universal gravitational constant  (m^3 kg^-1 s^-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity()
    {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double otherMass)
    {
        return otherMass * surfaceGravity();
    }

public static void main(String[] args)
{
   double earthWeight = Double.parseDouble(args[0]);

   double mass = earthWeight/EARTH.surfaceGravity();

   for (Planet p : Planet.values())
     System.out.printf("Your weight on %s is %f%n", p,
p.surfaceWeight(mass));
}



}
```

Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
public static void main(String[] args)
{
   double earthWeight = Double.parseDouble(args[0]);

   double mass = earthWeight/Planet.EARTH.surfaceGravity();

   for (Planet p : Planet.values())
     System.out.printf("Your weight on %s is %f%n", p,
p.surfaceWeight(mass));
}


Note: If the main is defined in a different class,
EARTH.surfaceGravity() should be
Planet.EARTH.surfaceGravity();
```

(4) Switching on Enums

```
public enum Suit
{ CLUBS, DIAMONDS, HEARTS, SPADES }

class Test
{  public static void main (String [] args)
   {  Suit s = Suit.CLUBS;

      switch(s)
      {
        case CLUBS: System.out.println("C"); break;
        case DIAMONDS: System.out.println("D"); break;
        case HEARTS: System.out.println("H"); break;
        case SPADES: System.out.println("S");
      }
   }
}
```

Note: the case statement requires that you not preface each enumerated value with the enum name (For example, case Suit.CLUBS). In fact, it's a compilation error if you do.

(5) Iterating over Enums

```
public void listSuitValues
{
   Suit[] suitValues = Suit.values();
   for (Suit s: Suit.values())
   { System.out.println(s); }
}
```

7.2 Type Inquiry

1. The `instanceof` operator

- op1 instanceof op2 returns true if op1 is an instance of op2.
- The operator is commonly used before down-casting.

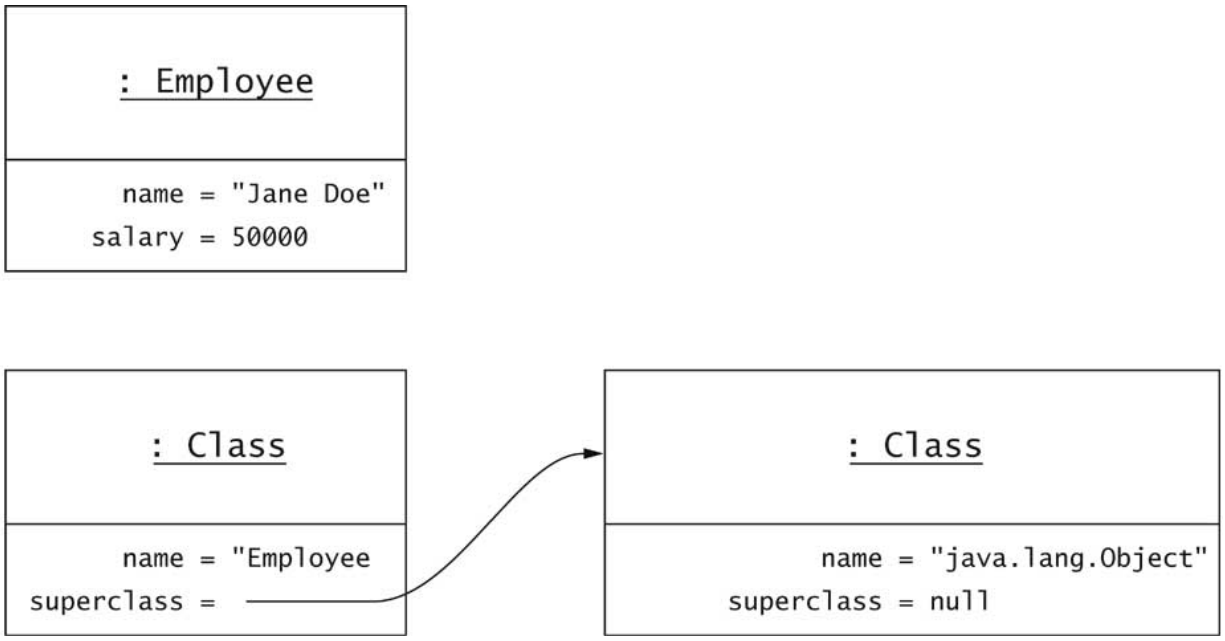
```
public void someMethod(Object x)
{
   if (x instanceof Shape)
   {
       Shape s = (Shape) x;
       g2.draw(s);

   }
}
```

- If e is null, test returns false (no exception is thrown.)

2. The Class Class

- Instances of the class Class represent classes and interfaces in a running Java application.
- Class has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the `defineClass` method in the class loader.
- A class object is a type descriptor. It contains information about a given type, such as the type name and the super class.
    - Example: An Employee Object vs. the Employee.class Object



- Useful public methods
  - The getClass method of the Object class

    returns the Class object that describes the objects' class.

    ```
    Object e = new Rectangle();
    Class c = e.getClass();
    ```

  - The `Class.getName` method returns the name of the entity (class, interface, array class, primitive type, or void) represented by this Class object, as a String

    ```
    System.out.println(c.getName()); // prints
                                     // java.awt.Rectangle
    ```

  - The Class.forName method yields Class object:

    ```
    Class c = Class.forName("java.awt.Rectangle");
    ```

- It is also possible to get the Class object for a named type (or for void) using a class literal, by applying a suffix .class to the type name.

```
Class c = Rectangle.class; //java.awt prefix not needed
if import java.awt.Rectangle is defined in the program.
```

- The Class class is a generic class with a type parameter. For example, Rectangle.class is an instance of Class<Rectangle>.

- The Class objects can describe any type including primitive type, class types, and interface types, and can also describe void.

Example: `int.class, void.class, Shape.class`

- There is only one Class object for every type that has been loaded into the Virtual machine. Therefore, it is safe to use == to test for equality.

  Example: Testing whether e is a Rectangle
  ```
  if (e.getClass() == Rectangle.class) ...
  ```

3. Array Type Inquiry

- Every array also belongs to a class that is reflected as a Class object that is shared by all arrays with the same element type and number of dimensions.

Example:
```
double[] a = new double[10];
double[] b = new double[5];
double[][] c = new double[2][3];
double[][] d = new double[5][2];

Class ca = a.getClass();
Class cb = b.getClass();
Class cc = c.getClass();
Class cd = d.getClass();

if (ca.isArray())
   System.out.println(ca.getComponentType()); // double

System.out.println(ca == cb); // true
System.out.println(cc == cd); // true
```

- The `getName` method produces strange names for array types

```
“[D” for double[]
“[[Ljava.lang.String;” for String[][]
```

7.3 Object: The Cosmic Superclass

- All classes extend `Object`
- Most useful methods defined in the Object class:
  - `String toString()`
  - `boolean equals(Object otherObject)`
  - `Object clone()`
  - `int hashCode()`

- Because only primitive functions are implemented in these functions, a class usually overrides them to supply useful functions.

1. The toString Method

- Returns a string representation of the object

Example: `Rectangle.toString` returns something like

```
java.awt.Rectangle[x=5,y=10,width=20,height=30]
```

- There are three cases where the `toString` method is automatically applied:
  - When you concatenate an object with a string

    `aString + anObject` means `aString + anObject.toString()`

  - When you print an object with the print or println method of the `PrintStream` and `PrintWriter` classes System.out.print(r);
  - When you pass an object reference e to an assert statement of the form

    ```
    assert condition : e;
    ```

(1) `Object.toString` prints class name and object address.

- Example: The GeneralPath class didn't override toString, so printing a GeneralPath object yields a printout something like this: java.awt.geom.GeneralPath@4abc9

(2) Overriding toString method

- Format all fields:

```
public class Employee
{   public String toString()
    {   return this.getClass().getName() + "[name=" + name +
",salary=" + salary + "]";  } ...
}
```

Typical string: `Employee[name=Harry Hacker,salary=35000]`

- Format super class first

```
public class Manager extends Employee
{   public String toString()
    {    return super.toString() + "[department=" + department + "]"; }
...
```

```
}
```

Typical string: `Manager[name=Dolly Dollar,salary=100000][department=Finance]`

2. The equals Method

`equals` tests for equal *contents* vs. == tests for equal *location*

Used many standard library classes such as `ArrayList.indexOf`

```
/**
   Searches for the first occurrence of the given argument,
   testing for equality using the equals method.
   @param elem an object.
   @return the index of the first occurrence
   of the argument in this list; returns -1 if
   the object is not found.
*/
public int indexOf(Object elem)
{
   if (elem == null)
   {
      for (int i = 0; i < size; i++)
         if (elementData[i] == null) return i;
   }
   else
   {
      for (int i = 0; i < size; i++)
         if (elem.equals(elementData[i])) return i;
   }
   return -1;
}
```

(1) Object.equals simply tests for identity

```
public class Object
{
   public boolean equals(Object obj)
   {
      return this == obj;
   }
   ...
}
```

(2) Overriding the equals Method

Basic rules

- Must cast the Object parameter to subclass
- Use == for primitive types, equals for object fields
- Call equals on super class

Example:

```
public class Manager extends Employee
{   public boolean equals (Object otherObject)
     {    if (!super.equals(otherObject)) return false;
          Manager other = (Manager) otherObject;
          return bonus == other.bonus;
     }
}
```

- Requirements for equals Method
  - *reflexive*: x.equals(x)
  - *symmetric*: x.equals(y) if and only if y.equals(x)
  - *transitive*: if x.equals(y) and y.equals(z), then x.equals(z)
  - x.equals(null) must return false

- Example that violates symmetry:

```
class Employee
{
   private String name;
   private int salary;

   public Employee(String n, int s)
   { name = n;
     salary = s;
   }
   public boolean equals(Object otherObject)
   {   if (otherObject instanceof Employee)
       {   Employee other = (Employee) otherObject;
           return name.equals(other.name) && salary == other.salary;
       }
       return false;
   }
}
class Manager extends Employee
{   private int bonus;
    public Manager(String n, int s, int b)
    { super(n,s);
      bonus = b;
    }
    public boolean equals (Object otherObject)
    {   if (!super.equals(otherObject)) return false;
        if (otherObject instanceof  Manager)
        {   Manager other = (Manager) otherObject;
```

```
            return bonus == other.bonus;
         }
         return false;
      }

}
public class Equals
{  public static void main (String [] args)
   { Employee e = new Employee("Smith", 10000);
     Manager m =  new Manager("Smith", 10000, 500);
     System.out.println(e.equals(null)); // null is not instanceof
Employee
     System.out.println(m.equals(null)); // null is not instanceof
Manager
     System.out.println(e.equals(m)); // true
     System.out.println(m.equals(e)); // false

   }
}
```

- The symmetry is violated due to the use of instanceof in the equals method (`m.equals(e)` returns false because e is not a Manager while e.equals(m) returns true because m is an Employee.)
- How to fix it ?
  - Add test for null: `if (otherObject == null) return false`
  - Use getClass method to test for class equality:

```
if (getClass() != otherObject.getClass()) return false;
```

- Therefore, an equals method should start out like this:

```
public boolean equals(Object otherObject)
{
   if (this == otherObject) return true; //reflexsive
   if (otherObject == null) return false; // test for null
   if (getClass() != otherObject.getClass())
      return false; //symmetry
   ...
}
```

Example:

```
class Employee
{   public boolean equals(Object otherObject)
     {   if (this == otherObject) return true; //reflexsive
         if (otherObject == null) return false; // test for
null
         if (getClass() != otherObject.getClass()) return
false; //symmetry
```

```
          Employee other = (Employee) otherObject;
          return name.equals(other.name) && salary ==
   other.salary;
        }
   }
   class Manager extends Employee
   {   public boolean equals (Object otherObject)
       {     if (this == otherObject) return true; //reflexsive
             if (otherObject == null) return false; // test for
   null
             if (getClass() != otherObject.getClass()) return
   false; //symmetry
             if (!super.equals(otherObject)) return false;

             Manager other = (Manager) otherObject;
             return bonus == other.bonus;
        }
   }
```

- Notion of equality depends on the abstract data type. For example, two sets can be considered equal if they contain the same elements in some order, not necessarily the same order, as for the equals method of the AbstractSet class.

// Let's assume that the Bag ADT implements the Set interface which doesn't store any duplicate elements.

```
   import java.util.*;
   class Bag<E>
   {  private ArrayList<E> data = new ArrayList<E>();


      // public methods of Bag are assumed to be here.

      public boolean equals(Object obj)
      {    if (this==obj) return true; // reflexive
           if (obj == null) return false; // test for null
           if (this.getClass() != obj.getClass()) return false;
   // symmetric

          Bag pbag = (Bag)obj;
          if (size() != pbag.size()) return false;

             // checks every element of this bag is contained in
the parameter bag
          boolean contains = false;

          for (int i=0; i < size(); i++)
          {   int j = 0; contains = false;
              while (!contains && j < pbag.size())
              { if (get(i).equals(pbag.get(j))) contains = true;
```

```
            else j++;
          }
          if (!contains) return contains;
       }
       return contains;
   }
   public static void main(String[] args)
   {
      Bag<String> b1 = new Bag<String>();
      Bag<String> b2 = new Bag<String>();
      b1.add("A"); b1.add("B");
      b2.add("B"); b2.add("A");
      System.out.println(b1.equals(b2)); //true
   }
}
```

3. The hashCode method

- Returns a hash code value for the object.

Used in a hash function index = objectname.hashCode()%table.length

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. if x.equals(y), then x.hashCode() == y.hashCode()

Example: hash code of String

```
int h = 0;
for (int i = 0; i < s.length(); i++)
    h = 31 * h + s.charAt(i);
```

Hash codes of "eat" and "tea" are 100184 and 114704, respectively.

(1) Object.hashCode() hashes the memory address of the object

(2) Overriding the hashCode method

- It is best to multiply individual hash codes with relatively prime factors before adding them together, to minimize the risk of collisions.

```
public class Employee
{  public int hashCode()
   {  return 11 * name.hashCode() + 13 *  new
```

```
Double(salary).hashCode(); }
   ...
}
```

- Two sets that are equal must yield the same hash code, even if the order of their elements differs.

```
public class AbstractSet ...
{   public int hashCode()
     {  int h = 0 ;
         Iterator i  = iterator();
         while (i.hasNext(h))
         {    Object obj = i.next();
              if (obj != null) h += obj.hashCode();
          }
          return h;
         }
}
```

7.4 Shallow and Deep Copy

A clone is an independent copy from the original.
Shallow copy: If a clone and the original share any *mutable* object, the clone is a shallow copy.

- Assignment (`copy = e`) does not make a clone.
- General contracts of a clone method
  - `x.clone() != x`
  - `x.clone().equals(x)`
  - `x.clone().getClass() == x.getClass()`

(1) `Object.clone` makes a shallow copy

The default implementation assigns each field from the source to the same field in the destination object. Therefore, if a filed contains a reference, the reference is copied to the same field in the cloned object resulting two fields share the same object.

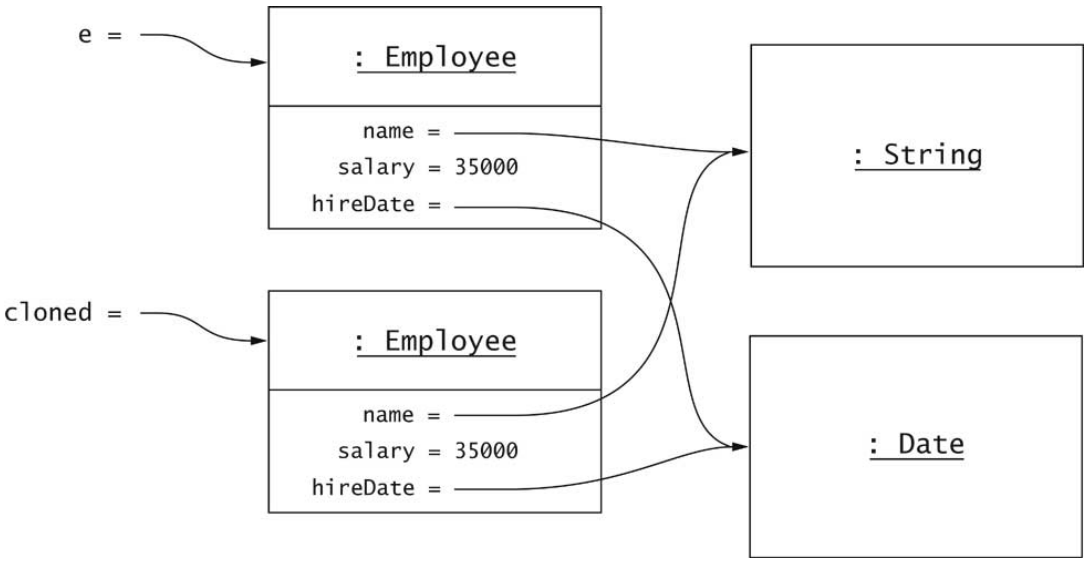
(2) Overriding the `clone` method

- Three important factors in writing a clone method:
  - Object.clone checks whether the object on which it was invoked implements the Cloneable interface and throws CloneNotSupportedException if it does not.
  - The CloneNotSupportedException is a checked exception. It must be handled by either try-catch or throws clause.
  - Object.clone is protected. Therefore, if a class wants to allow clients to clone its instances, it must redefine clone to a public method.

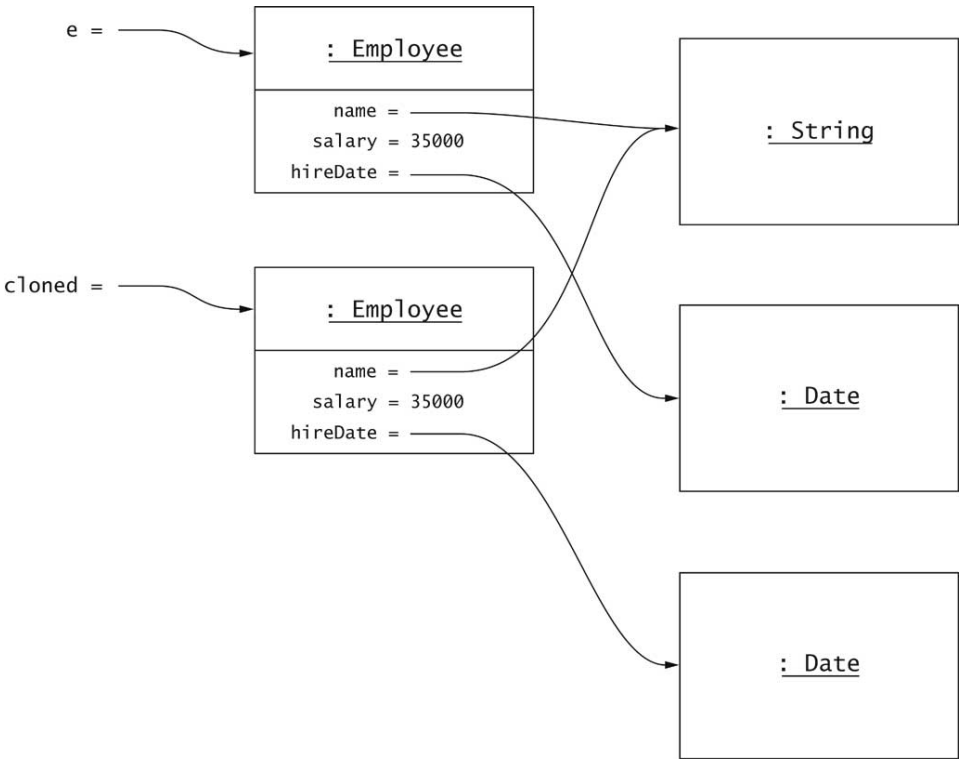
- Example

```
public class Employee implements Cloneable
{ public Employee clone() //throws
CloneNotSupportedException
  {  try
     {
         Employee cloned = (Employee)super.clone();
         cloned.hireDate = (Date)hiredate.clone();
         return cloned;
      }
      catch(CloneNotSupportedException e)
     { return null; // won't happen }
}
```

Shallow copy done by super.clone()



Deep copy



7.5 Serialization

7.5.1 Why Serialization ?

With object *serialization*, a program can read/write a whole object to/from a raw byte stream. That is a whole object is encoded into a raw byte stream that is suitable for streaming to a network or to a file-system. The *deserialization* is a process to reconstitute an object from the byte stream.

7.5.2. How to serialize/deserialize an object ?

1. The class of an object must implement the `Serializable` interface (`java.io.Serializable`) which has no methods that you need to write.

```
public class Employee implements Serializable
{}
```

2. To serialize an object, use `ObjectOutputStream`

```
Employee[] staff = new Employee[2];
staff.add(new Employee(...));
staff.add(new Employee(...));
```

```
ObjectOutputStream out =
    new ObjectOutputStream(new
FileOutputStream("staff.data"));
out.writeObject(staff);
out.close();
```

Java arrays and Employees are serializable. The above code saves an array of Employee objects, the array itself and all objects that it references, to the file.

3. To deserialize the complete object, use `ObjectInputStream`

```
ObjectInputStream in =
   new ObjectInputStream(new FileInputStream("staff.data");
Employee[] staff = (Employee[]) in.readObject();
in.close();
```

7.5.3 How serialization works

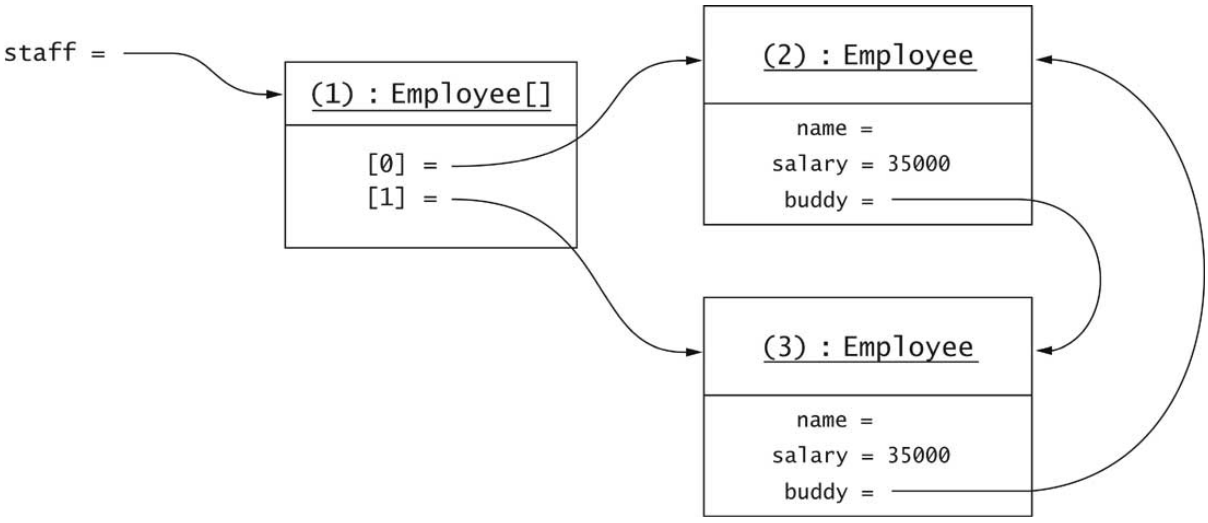
Consider an Employee class with fields name, salary, and buddy that references another Employee.

```
public class Employee implements Serializable
{    private String name;
     private double salary;
     private Emplyee buddy;

     …

}
```

- Each newly encountered object is saved
- Each object gets a serial number in the stream
- When an object is previously saved, only the serial number is saved.
    - No object is saved twice
    - Reference to already encountered object saved as "reference to #"



- Object#1, type = Employee[]
  - [0] component is Object #2, type = Employee
    - name field
    - salary field
    - buddy field is Object #3, type = Employee
      - name field
      - salary field
      - buddy field is Object #2 (already saved)
  - [1] component is Object #3 (already saved)

3. transient field

There are two different purposes of using a transient field.

1. To protect sensitive information and functions

```
private transient boolean selected;
```

2. To avoid errors with instance fields of types that are not serializable such as Ellipse2D.Double.

```
private transient Ellipse2D frontTire;
```

3. Example: Car.java

Ch7/serial/Car.java

- Supply private methods
  - private void writeObject(ObjectOutputStream out)
  - private void readObject(ObjectInputStream in)
- In these methods,

- Call writeDefaultObject/readDefaultObject (write/read non-static and non-transient fields of the class to the stream)
- Manually save other data

```
public class SerializeCarTester
{
   public static void main(String[] args)  throws IOException, ClassNotFoundException
   {   Car beemer = new Car(100, 100, 60);
       System.out.println(beemer);
       ObjectOutputStream out
             = new ObjectOutputStream(new FileOutputStream("fleet.dat"));
       out.writeObject(beemer);
       out.close();

       ObjectInputStream in
             = new ObjectInputStream(new FileInputStream("fleet.dat"));
       System.out.println(in.readObject());
       in.close();
   }
}

import java.awt.*;
import java.awt.geom.*;
import java.io.*;

/**
    A serializable car shape. Refer to the text book for the completed class.
*/
public class Car implements Serializable
{
    private Rectangle body;
    private Rectangle roof;
    private transient Ellipse2D.Double frontTire;
    private transient Ellipse2D.Double rearTire;

   private void writeObject(ObjectOutputStream out) throws IOException
   {
     out.defaultWriteObject();
     writeRectangularShape(out, frontTire);
     writeRectangularShape(out, rearTire);
   }

   /**
      A helper method to write a rectangular shape.
      @param out the stream onto which to write the shape
      @param s the shape to write
```

```
      */
      private static void writeRectangularShape(ObjectOutputStream out, RectangularShape s)
                                                                        throws IOException
      {
        out.writeDouble(s.getX());
        out.writeDouble(s.getY());
        out.writeDouble(s.getWidth());
        out.writeDouble(s.getHeight());
      }

      private void readObject(ObjectInputStream in)
                                          throws IOException, ClassNotFoundException
      {
        in.defaultReadObject();
        frontTire = new Ellipse2D.Double();
        readRectangularShape(in, frontTire);
        rearTire = new Ellipse2D.Double();
        readRectangularShape(in, rearTire);
      }

      /**
        A helper method to read a rectangular shape.
        @param in the stream from which to read the shape
        @param s the shape to read. The method sets the frame
        of this rectangular shape.
      */
      private static void readRectangularShape(ObjectInputStream in, RectangularShape s)
                                                                        throws IOException
      {
        double x = in.readDouble();
        double y = in.readDouble();
        double width = in.readDouble();
        double height = in.readDouble();
        s.setFrame(x, y, width, height);
      }


}
```

Interesting Tip: Who is the calling method (caller) ?

```
private void writeObject(ObjectOutputStream out) throws IOException
{
    try
    {   throw new Exception("Who called me?"); }
    catch(Exception e)
    {     System.out.println("I was called by ");
```

```
            for(StackTraceElement s: e.getStackTrace())
            { System.out.println(s.getClassName() + "." + s.getMethodName());
            }
        }
        out.defaultWriteObject();
        writeRectangularShape(out, frontTire);
        writeRectangularShape(out, rearTire);
    }
}
```

Change the method as shown above and run the program. You will find that the private writeObject method is initially called by the ObjectOutputStream.writeObject method.

7. 6 Reflection

Java reflection is useful because it supports dynamic retrieval of information about classes and data structures by name, and allows for their manipulation within an executing Java program.

`java.lang.reflect`

`Class`	Describes a type
`Package`	Describes a package
`Field`	Describes a field and allows inspection and modification of fields
`Method`	Describes a method and allows its invocation on objects
`Constructor`	Describes a constructor and allows its invocation
`Array`	Has static methods to analyze arrays

7.6.1 Setting Up to Use Reflection

There are three steps that must be followed to use these classes. Suppose you want to inspect methods.

1. To obtain a java.lang.Class object for the class that you want to manipulate. There are several ways to obtain a Class object.

   - `Class c = objectName.getClass();`

   The method returns the Class object created when the class was loaded.

- `Class c = Class.forName("java.awt.Rectangle");`

  Returns the Class object associated with the class or interface with the given string name. If the class or interface hasn't been loaded, the method attempts to locate and load the class. If the class cannot be located, ClassNotFoundException is thrown.

- `Class c = java.awt.Rectangle.class;`

  This approach is equivalent to using Class.forName().

2. To call a method such as `getDeclaredMethods`, to get a list of all the methods declared by the class.

3. To use the reflection API to manipulate the information. For example, the sequence:

```
Class c = Class.forName("java.lang.String");
Method[]  m= c.getDeclaredMethods();
System.out.println(m[0].toString()); // returns a string describing this method.
```

7.6.2 The Class class

- A reflection starts with a Class object. From the Class object you can obtain
  - complete list of members of the class
    - names and types of fields
    - names, parameter types, return types of methods
    - parameter types of constructors
  - all the types of the class
    - interfaces it implements
    - superclasses it extends
  - information about the class itself such as
    - modifiers applied to it (public, abstract, final, and so on)
    - package it is contained in.

- Useful methods for enumerating (Refer to Java API to find more functions.)
  - `Class getSuperclass()`
  - `Class[] getInterfaces()`
  - `Package getPackage()`
  - `Field[] getDeclaredFields()`
  - `Constructor[] getDeclaredConstructors()`
  - `Method[] getDeclaredMethods()`
- Example: The program takes a class name as a command line argument, and prints the class name, the name of its super class, and the name of its public methods.

```
import java.lang.reflect.*;
public class SimpleClassDesc
```

```
{   public static void main (String[] args)
   {  Class type = null;
      try
      {   type = Class.forName(args[0]); }
      catch (ClassNotFoundException e)
      {   System.err.println(e);
         return;
      }

      System.out.print("class " + type.getSimpleName());
      Class superclass = type.getSuperclass();

      if (superclass != null)
          System.out.println(" extends " + superclass.getCanonicalName());
      else
          System.out.println();
      Method[] methods = type.getDeclaredMethods();
      for (Method m : methods)
          if (Modifier.isPublic(m.getModifiers()))
              System.out.println(" " + m);
  }
}
```

7.6.3 The Constructor class: Constructor Inspection

```
Constructor[] cons = Rectangle.class.getDeclaredConstructors();
for (Constructor c : cons)
{   Class[] params = c.getParameterTypes();
    System.out.print("Rectangle(");
    boolean first = true;
    for (Class p : params)
    {
       if (first) first = false; else System.out.print(", ");
       System.out.print(p.getName());
    }
    System.out.println(")");
}
Output
Rectangle()
Rectangle(java.awt.Rectangle)
Rectangle(int, int, int, int)
Rectangle(int, int)
Rectangle(java.awt.Point, java.awt.Dimension)
Rectangle(java.awt.Point)
Rectangle(java.awt.Dimension)
```

7.6.4 The Method class: Invoking a Method

- The Method class provides information about, and access to, a single method on a class or interface. The reflected method may be a class method or an instance method (including an abstract method).
- The invoke method of the Method class invokes the underlying method represented by this Method object, on the specified object with the specified parameters.

```
public Object invoke(Object obj,Object... args)
```

Example: To call `System.out.println("Hello, World")` the hard way.

```
Method m = PrintStream.class.getDeclaredMethod("println",String.class);
m.invoke(System.out, "Hello, World!");
```

- To get a declared method without a parameter

```
Method m =java.awt.Rectangle.class.getDeclaredMethod("toString",null);
```

- To invoke a method without a parameter

```
System.out.println(m.invoke(new Rectangle(10,10,20,20), null));
// prints java.awt.Rectangle[x=10,y=10,width=20,height=20]
```

- To invoke a static method

```
Method m = Math.class.getDeclaredMethod("sqrt", double.class);
double r = (Double) m.invoke (null, 4.0); // Explicit type casting
is required to convert Object to Double; The Double object is auto-
unboxed by Java 5 to a double number.
```

- If the method returns a value, the invoke method returns it as an Object. If the value has a primitive type, it is first appropriately wrapped in an object.
- However, if the value has the type of an array of a primitive type, the elements of the array are *not* wrapped in objects; in other words, an array of primitive type is returned.
- If the return type of the invoked method is void, the invocation returns null.

7.6.5 The Field class: Inspecting Objects

- To dynamically look up the fields of objects at runtime.

- Example: To print the names of all static fields of the Math class.

```
Field[] fields = Math.class.getDeclaredFields();
for (Field f : fields)
   if (Modifier.isStatic(f.getModifiers()))
      System.out.println(f.getName());
```

- Example: To inspect a private field of an object

```
class BankAccount
{ private double balance;
  public double getBalance()
  { return balance; }
}

BankAccount b = new BankAccount();
Class c = b.getClass();
Field f = c.getDeclaredField("balance");// returns a Field object
that reflects the specified field.

f.setAccessible(true);  // see the discussion below.
Object value = f.get(b); //Returns the value of the specified field
                         //of the object b.
f.set(b, 100); //Sets the specified field of the object b to the new value.
System.out.println(b.getBalance());  //prints 100
```

Note: The `setAccessible` call can be protected by installing a security manager. By default, Java applications do not install a security manager.

- Example: Peek inside a randomizer.

```
public class FieldTester
{
  public static void main(String[] args) throws IllegalAccessException
  {
    Random r = new Random();
    System.out.print(spyFields(r));
    r.nextInt();
    System.out.println("\nAfter calling nextInt:\n");
    System.out.print(spyFields(r));
  }

  public static String spyFields(Object obj)  throws IllegalAccessException
  {
    StringBuffer buffer = new StringBuffer();
    Field[] fields = obj.getClass().getDeclaredFields();
    for (Field f : fields)
    {
      if (!Modifier.isStatic(f.getModifiers()))
      { //  buffer.appends to format the output are omitted here.
        f.setAccessible(true);
        Object value = f.get(obj);
        buffer.append(f.getType().getName());
        buffer.append(f.getName());
        buffer.append("" + value);
```

```
        }
      }
      return buffer.toString();
    }
  }
```

```
java.util.concurrent.atomic.AtomicLong seed=28868614649646
double nextNextGaussian=0.0
boolean haveNextNextGaussian=false

After calling nextInt:

java.util.concurrent.atomic.AtomicLong seed=183339123151521
double nextNextGaussian=0.0
boolean haveNextNextGaussian=false
```

7.6.6 The Array class: Inspecting Array Elements

- You can inspect an array element using static methods of Array class. With an array object a,

Object value = Array.get(a,i); // reads a value at index i
Array.set(a, i, value); // sets a value of an element at index I
Array.getLength(a); // gets the length of the array

- Example: To double the size of the array using reflection.

```
double[] a = {1,2,3,4};
Object anew =
Array.newInstance(a.getClass().getComponentType(),2*Array.getLength(a));
System.arraycopy(a, 0, anew, 0, Array.getLength(a));
a = (double[])anew;
for(double d: a) { System.out.print(d + " "); }
```

7.7 Generic Types

1. Generic Type Declarations

- A *generic type declaration* has one or more type parameters.

```
public class ArrayList<E>
{   public E get(int i) { . . . }
    public E set(int i, E newValue) { . . . }
      . . .
    private E[] elementData;
}
```

The ArrayList is declared as ArrayList<E>. E represents the type of element that an array list object can hold. E is known as a type parameter for which a concrete type argument can be substituted. By convention, type variables have a single character names: E for an element type, K for a key type, V for a value type, T for a general type.

- In a *generic type invocation*, concrete type arguments are substituted for the formal type parameters.

Example of a generic type invocation

```
ArrayList<String> myList = new ArrayList<String>();
myList.add("hello"); // compiles
String x = myList.get(0); // no type casting required.
```

- A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration. The use of <String> is for the compiler to check that the object is used correctly. At run time, no generic type information is present in objects. There is only one class, so called *raw* class.
- Cannot use primitive types, e.g. no `ArrayList<int>`

2. Bounded Type Parameters

- In the above generic class declaration of ArrayList<E>, the type variable E can be replaced by any reference type. Sometimes you may want to restrict the type parameters of a generic class.

```
interface SortedCollection<E extends Comparable<E>>
{  // ... sorted collection methods ...

}
```

- Here, E is restricted to be a type that "extends" Comparable<E> so that it is guaranteed to support the methods of the Comparable interface. The Comparable is the *upper* bound on the type of E.
- The keyword extends is used to mean either "extends" or "implements" depending on whether the type that follows is a class type or an interface type.
- Multiple dependencies can be expressed by declaring that the type parameter extends one class or interface, followed by & separated list of additional interfaces.

```
interface SortedCharSeqCollection
                < E extends Comparable<E> & CharSequence> {
{ // ... sorted char sequence collection methods ...
}
```

3. Subtyping and Wildcards

(1) Subtyping of non-wildcard version

- If S is a subtype of T, ArrayList<S> is *not* a subtype of ArrayList<T>.

  Example 1

```
public static void method (List<Object> lo)
{   lo.add(new Object()) ; }

method (new ArrayList<Object>()); // yes
method (new ArrayList<String>()); // compilation error
```

  Example 2

```
static double sum (List<Number> list)
{
    double sum = 0.0;
    for (Number n : list)
       sum += n.doubleValue();
    return sum;
}

List<Integer> data = new ArrayList<Integer>(); // yes
data.add(1);
data.add(4);
double sum = sum(data); // compilation error
```

- The meaning of the parameter List<Number>
  - (correct) an object that is compatible with `List` that has elements declared to be Number.
  - (incorrect) an List object that has elements that are compatible with `Number`.

- Therefore, if you try to invoke sum with a List<Integer>, the compiler will complain.
  Reason of the error: Although Integer is a subtype of Number, List<Integer> is not a subtype of List<Number>. (This is quite contrast with arrays, where Integer[] is a subtype of Number[].)

(2) Sub-typing of bounded wildcard version

- The solution to the problem is to use of *bounded wildcard*, where Number forms the upper bound on the expected type.

```
static double sum (List<? extends Number> list)
{ double sum = 0.0;
```

```
    for (Number n : list)
         sum += n.doubleValue();
    return sum;
}
```

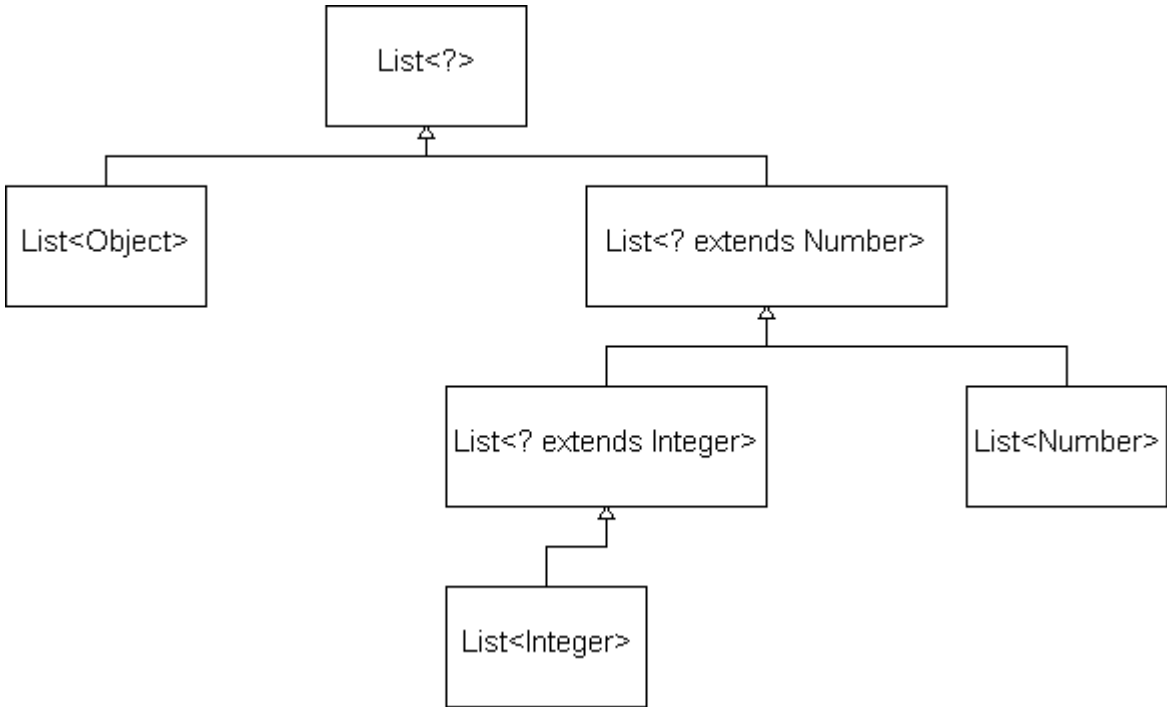
- Any question ?

[Q1] Can I set the lower bound of the type variable ?
[A1] Yes. Use the keyword `super` rather than extends. For example, List<? super Integer> means a List of Integer or any of its super types such as List<Integer>, List<Number>, List<Comparable<Integer>>, or List<Object>.

[Q2] Can a bounded wildcard have multiple bounds ?
[A2] No. Unlike a bounded type variable (in a generic type declaration), a bounded wildcard can have only a single bound - either an upper bound or a lower bound. For example, List<? extends Value & Serializable> is an error.

[Q3] Can I use an unbounded wild card ?
[A3] Yes. List<?> represents a list of any kind - the upper bound is implicitly Object. It is important to note that List<?> is not equivalent to List<Object>; rather it means List<? extends Object> .

- Relationship between bounded wildcard types



- Reasonable restriction on unbounded or upper-bounded wild card type

" A wildcard represents an unknown type, therefore, you can't do anything that requires the type to be known."

Example 1

```
public void someMethod(SingleLinkQueue<?> anyqueue)
{  anyqueue.add("Hello"); // won't compile}
```

Reason of the error: The compiler finds a method to call based on the type SingleLinkQueue<?> which can be a queue for any objects (It doesn't have any idea what specific queue will be assigned to anyqueue.) Therefore, if you activate a method to add a String on queue, the compiler complains because it simply cannot guarantee a String queue.

Example 2

```
public void someMethod(SingleLinkQueue<? extends Number> numbers)
{  numbers.add(new Integer(25)); // won't compile }
```

Reason of the error: To the compiler, the parameter `numbers` can be a queue that stores Number objects or objects of a subclass of Number. If you activate a method to add an Integer object, the compiler complains. It is not determined at the compilation level if the actual storage will be for Number objects or Integers.

- What about a lower-bound wildcard type ? The following is perfectly correct:

```
static void addString(SingleLinkQueue<? super String> sq)
{ sq.add("Hello"); } // why ?
```

4. Generic Methods and Constructors

(1) Overview

- Generic methods and constructors are typically used when you need 1) to introduce a type variable to constrain the parameterized types of different parameters, or 2) to constrain a parameter and the return type.
  (Otherwise, don't use a generic method.)
- A generic method can be declared inside an ordinary class or a generic class. You declare a generic method by defining type variables between the method modifiers and the method return type.

```
public class Utils
{
   public static  <E> void fill(ArrayList<E> a, E value, int count)
   {
```

```
         for (int i = 0; i < count; i++) a.add(value);
      }
   }
```

- Example of bounded type parameter in a generic method definition

```
public static <E extends Comparable<? super E>> E
getMax(ArrayList<E> a)
{
      E max = a.get(0);
      for (int i = 1; i < a.size(); i++)
         if (a.get(i).compareTo(max) > 0) max = a.get(i);
      return max;
}
```

[Q] What if the method is defined as follows ?

```
public static <E extends Comparable <E>> E getMax(ArrayList<E> a)

public abstract class Calendar implements Comparable<Calendar> { … }
public class GregorianCalendar extends Calendar { }
```

[A] When a method is called with ArrayList<GregorianCalendar>, the compiler complains, because GregorianCalendar (indirectly) implements Comparable<Calendar>, not exactly Comparable<GregorianCalendar> as the bounded type parameter requires.

```
public class GenericMethod
{
     public static <E extends Comparable<E>> E
getMax(ArrayList<E> a)
     { E max = a.get(0);
       for (int i = 1; i < a.size(); i++)
         if (a.get(i).compareTo(max)>0) max = a.get(i);
                       return max;
     }
     public static void main (String [] args)
     {
          ArrayList<GregorianCalendar> data = new
ArrayList<GregorianCalendar>();
          data.add(new GregorianCalendar());
          data.add(new GregorianCalendar());
          System.out.println(getMax(data)); // compilation error
     }

}
```

[Q] When should I use generic methods, and when should I use wildcard types ?

```
interface Collection<E>
{
public boolean containsAll(Collection<?> c);
public boolean addAll(Collection<? extends E> c);
}
```

We could have used generic methods here instead:

```
interface Collection<E>
{
public <T> boolean containsAll(Collection<T> c);
public <T extends E> boolean addAll(Collection<T> c);
}
```

[A] Generic methods allow type parameters to be used to express dependencies among the types of one or more parameters to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

(2) Generic Method Invocation and Type Inference

- You can parameterize a method invocation to supply type arguments for the methods' type variables.

```
public <T> T passThrough(T obj) { return obj; }

String s1 = "Hello";
String s2 = this.<String>passThrough(s1);
```

- In the absence of a type argument, the compiler will infer what type to use from the static argument types and the way in which the return type is used.

```
String s1 = "Hello";
String s2 = passThrough(s1);
Object o1 = passThrough(s1); // T => String
Object o2 = passThrough((Object) s1);  // T => Object
s1 = passThrough((Object) s1);
// won't compile. s1 = (String) passThrough((Object) s1);
```

- Note: If you make a parameterized method invocation, you must qualify the method name appropriately, such as by using this or super for instance methods, or the class name for static methods.

```
String s1 = “Hello”;
String s2 = <String>passThrough(s1); // INVALID
```

- More examples of type inference

```
public class Utils
{
```

```
   public static <E> void fill(ArrayList<E> a, E value, int
   count)
   {
      for (int i = 0; i < count; i++) a.add(i,value);
   }
}
```

Example 1

```
ArrayList<String> ids = new ArrayList<String>();
Utils.fill(ids, "default", 10);
```
(ArrayList<E> and E) against (ArrayList<String> and String)

Example 2

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
Utils.fill(shapes, new Rectangle(5,10,20,30), 10);
```
(ArrayList<E> and E) against (ArrayList<Shape> and Rectangle): E→ Shape

Example 3: Better to Specify the instantiation for clarity.

```
Utils.<Shape>fill(shapes, new Rectangle(5,10,20,30),10);
```

5. Erasure and Raw types

- The compiler essentially erases all generic type information from the compiled class. The erasure of a generic type is known as the *raw type*. For example, the erasure of Cell<E> is just Cell.
- The erasure of a type variable is the erasure of its first bound.
  - `<E> --> Object`
  - `<E extends Number> --> Number`
  - `<E extends Number & Cloneable> --> Number`

- Example: `ArrayList<E>` becomes

```
public class ArrayList
{
   public Object get(int i) { . . . }
   public Object set(int i, Object newValue) { . . . }
   . . .
   private Object[] elementData;
}
```

- When the type information from the erasure of the generic type doesn't match what is expected, the compiler inserts a cast.

```
ArrayList<String> a = new ArrayList<String>();
```

```
a.add("Hello");
String s = q.get(0);   // the compiler inserts a cast to String
                       // because the erasure of type variable in
                       // ArrayList<E> is just Object.
```

- Example

```
public static <E extends Comparable<? super E>> E getMax(ArrayList<E>
a)
public static Comparable getMax(ArrayList a)
 // E extends Comparable<? super E> erased to Comparable
```

- Erasure necessary to interoperate with legacy (pre-JDK 5.0) code

6. Limitations of Generics

- Cannot replace type variables with primitive types
- Cannot construct new objects of generic type

```
public static <E> void fillWithDefaults(ArrayList <E> a,
int count)
{   for (int i= 0; i < count; i++)
      a.add(new E()); // Error
}
```

After the type erasure process, the above code becomes

```
public void fillWithDefaults(ArrayList  a, int count)
{   for (int i= 0; i < count; i++)
         a.add(new Object());
}
```

- You can declare array *types* whose element type is a type variable. However, generic array creation is not allowed.

```
public class ArrayList<E>
{
   private E[] theData; // ok

   public ArrayList()
   {
      // theData = new E[capacity]; error
      theData = (E[]) new Object[capacity];
   }
}
```

- Cannot reference type parameters in a static context

```
public class MyClass<E>

{
   private static E defaultValue; // Error
}
```

- Cannot throw or catch generic types. In fact, a generic type cannot extend Throwable.
- Cannot have type clashes after erasure. Ex. `GregorianCalendar` cannot implement `Comparable<GregorianCalendar>` since it already implements `Comparable<Calendar>`, and both erase to `Comparable`