

4. Class Hierarchies

## Class Hierarchies

### Subclass Definition

```

class Person {
    String name;
    int age;
    public String toString () {
        return "Name: " + name + "\nAge: " + age;
    }
}

class Employee extends Person {
    long grossSalary;
    Manager supervisor;
}

class Manager extends Employee {
    int category;
    ArrayList subordinates = new ArrayList ();
    void promote () { category++; }
}
    
```

2

### Type Hierarchy (I)

```

Person x1, x2;
Employee y, emp = new Employee ();
Manager z, mgr = new Manager ();
    
```

- Implicit, automatic conversion (generalization)
  - x1 = emp; // Employee → Person *A Manager can automatically play the role of Employee and Person*
  - x2 = mgr; // Manager → Person
  - y = mgr; // Manager → Employee
- Explicit conversion (specialization), programmer responsibility *A Person can play the role of Manager if it is really a Manager*
  - z = x2; // Compile error
  - z = (Manager) x2; // Person → Manager
  - z = (Manager) x1; // Runtime error: x1 is not a Manager
  - z = (Dog) x1; // Compile error (unless Dog was a subclass or superclass of Person)

### Type Hierarchy (II)

```

class C {
    void f (Employee p) { ... }
    void g (Manager p) { ... }
}

Manager mgr = new Manager ();
Employee x = mgr, emp = new Employee ();
C c = new C ();
    
```

- Implicit conversion
  - c.f (mgr); // Manager → Employee
- Explicit conversion
  - c.g (x); // Compile error
  - c.g ((Manager) x); // Employee → Manager
  - c.g ((Manager) emp); // Runtime error: emp is not a Manager

4

### Variable and Method Inheritance

```

Employee emp = new Employee ();
Manager mgr = new Manager ();

emp.name = "Peter";
emp.age = 28;
emp.netSalary = 200;
emp.supervisor = mgr;
System.out.println (emp.toString ());

dir.name = "Mary";
dir.age = 45;
dir.grossSalary = 700;
dir.supervisor = null;
dir.category = 3;
System.out.println (dir.toString ());
    
```

5

### Inheritance and Type Hierarchy

```

Manager mgr = new Manager ();
Employee emp = mgr;
Person p = mgr;

p.grossSalary = 100; // Error: grossSalary not defined for Person
emp.promote (); // Error: promote not defined for Employee
    
```

Static vs. dynamic type

```

mgr → {
    emp → {
        p → {
            name
            age
            salary
            supervisor
            category
        }
    }
}
    
```

6

4. Class Hierarchies

### Variable and Method Overriding

- Redefinition of parent class variables and methods in a subclass
- The definition in the subclass overrides the one inherited from the superclass
- The definition in the superclass can be accessed from the subclass with **super**
- Method overriding (specialization)
  - The method is redefined with the same arguments and return type
  - If the type of arguments is not exactly the same, it is not overriding but overloading
  - The access permission of the overridden methods cannot be made more restrictive
  - Like method overloading, method overriding avoids the proliferation of identifiers
  - It enables dynamic linking
- Variable overriding
  - The variable gets a separate memory space in each class
  - The type does not need to be the same
  - Static linking
  - But it is discouraged in general to override variables

7

### Variable Overriding

```
class Musician extends Person {
    String name;
    void showNames () {
        System.out.println ("Musician: " + name);
        System.out.println ("Person: " + super.name);
    }
}

// Main block
Musician m = new Musician ();
Person p = m;
m.name = "Stevie Wonder"; // Musician name
p.name = "Stevland Morris"; // Person name
```

8

### Method Overriding

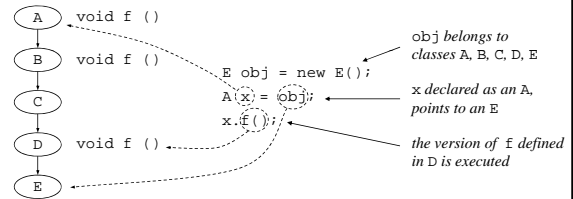
```
class Employee extends Person {
    long grossSalary;
    Manager supervisor;
    public String toString () {
        return "Name: " + name + "\nAge: " + age +
            "\nIncome: " + grossSalary + "\nSupervisor: " +
            ((supervisor == null)? name : supervisor.name);
    }
}

// Main block
Employee emp = new Employee ();
Person p = emp;
emp.toString (); // Employee version of toString
p.toString (); // Employee version of toString
```

9

### Dynamic Linking

- Overridden method calls are solved by dynamic linking at runtime
- The definition in the most specific class of the object is executed, regardless of how the object reference has been declared



- Static methods use static linking

10

### Dynamic Linking: Example

```
class Person {
    String name;
    int age;
    public String toString () {
        return "Name: " + name + "\nAge: " + age;
    }
}

class Employee extends Person {
    long grossSalary;
    Manager supervisor;
    public String toString () {
        return "Name: " + name + "\nAge: " + age +
            "\nIncome: " + grossSalary + "\nSupervisor: " +
            ((supervisor == null)? name : supervisor.name);
    }
}
```

11

```
class Manager extends Employee {
    int category;
    ArrayList subordinates = new ArrayList ();
    public String toString () {
        return "Name: " + name + "\nAge: " + age +
            "\nIncome: " + grossSalary + "\nSupervisor: " +
            ((supervisor == null)? name : supervisor.name) +
            "\nCategory: " + category;
    }
    void promote () { category++; }
}
```

12

4. Class Hierarchies

```

// Main block
Manager mgr = new Manager ();
Employee emp = new Employee ();
Employee z = mgr;
Person p = new Person ();
Person x = emp;
Person y = z;

p.toString (); // Person version of toString
emp.toString (); // Employee version of toString
dir.toString (); // Manager version of toString
x.toString (); // Employee version of toString
y.toString (); // Manager version of toString
z.toString (); // Manager version of toString
y.promote (); // ERROR
    
```

13

### Linking is Static with Respect to Arguments

```

class A {
void f (Person p) {
    System.out.println ("Class Person");
}
void f (Employee emp) {
    System.out.println ("Class Employee");
}
}
    
```

The most specific compatible definition is executed

```

// Main block
A a = new A ();
Manager mgr = new Manager (...);
Person p = mgr;
a.f (mgr);
a.f (p); // Assuming Person is subclass of Animal
Animal x = p;
a.f (x); // ERROR
    
```

14

## Inheritance and Constructors

Constructors are not inherited (and therefore cannot be overridden)

- When creating an Employee, the constructor of Person is invoked
- Implicit automatic invocation
  - The constructor of the parent class without arguments is invoked
  - If it is not defined → error
- Explicit invocation
  - Invocation to `super (...)` in the first line of the Employee constructor
- Invocation to other constructors of the same class: `this (...)`

15

## Inheritance and Constructors: Example

```

class Person {
String name;
int age;
Person (String str, int i) {
    name = str;
    age = i;
}
public String toString () {
    ...
}
}
// Error when creating an Employee: the default constructor
// Employee () invokes the default constructor Person (),
// which is no longer defined
    
```

16

```

class Employee extends Person {
long grossSalary;
Manager supervisor;
Employee (String str, int i, long salary, Manager mgr) {
    name = str;
    age = i;
    grossSalary = salary;
    supervisor = mgr;
}
public String toString () {
    ...
}
}
// Error when creating an Employee: Person () keeps being
// invoked by default
// Error when creating a Manager: the default constructor
// Manager () invokes the default constructor Employee (),
// which is no longer defined
    
```

17

```

class Employee extends Person {
long grossSalary;
Manager supervisor;
Employee (String str, int i, long salary, Manager mgr) {
    super (str, i);
    grossSalary = salary;
    supervisor = mgr;
}
public String toString () {
    ...
}
}
    
```

18

4. Class Hierarchies

```

class Manager extends Employee {
    int category;
    ArrayList subordinates = new ArrayList ();
    Manager (String str, int i, long n, int j) {
        this (str, i, n, null, j);
    }
    Manager (String str, int i, long n,
            Manager mgr, int j) {
        super (str, i, n, mgr);
        category = j;
    }
    public String toString () {
        ...
    }
    void promote () { category++; }
}
    
```

Always on the first line

19

### Access Control: private

```

class Person {
    private String name;
    private int age;
    public String toString () {
        return "Name: " + name + "\nAge: " + age;
    }
}

class Employee extends Person {
    long grossSalary;
    Manager supervisor;
    public String toString () {
        return "Name: " + name + "\nAge: " + age +
            "\nIncome: " + grossSalary + "\nSupervisor: " +
            ((supervisor == null)? name : supervisor.name);
    }
}
    
```

Error: name and age are private

20

### Access Control: protected (I)

```

class Person {
    protected String name;
    protected int age;
    protected String toString () {
        return "Name: " + name + "\nAge: " + age;
    }
}

class Employee extends Person {
    long grossSalary;
    Manager supervisor;
    String toString () {
        return super.toString () +
            "\nIncome: " + grossSalary + "\nSupervisor: " +
            ((supervisor == null)? name : supervisor.name);
    }
}
    
```

Error: can only be protected or public

Correct even if Employee and Person were in different packages

In different package would be an error if supervisor was Person but not Employee

21

### Access Control: protected (II)

```

package personal;

public class Person {
    protected String name;
    protected int age;
    protected String toString () {
        return "Name: " + name + "\nAge: " + age;
    }
}

package personal;
...
// In any class
Person p = new Person ();
p.toString ();
...

package X;
...
// In any class
personal.Person p =
    new personal.Person ();
p.toString (); // Error
...
    
```

### Final Classes and Final Members

```

final class A {
}

class B extends A { // Error: A cannot be subclassed
    final int x; // Error: x must be initialized
    final double f (int x) {
        return (x-1)/(x+1);
    }
}

class C extends B {
    int x;
    double f (int x) { // Error: f cannot be overridden
        return (x-2)/(x+2);
    }
}
    
```

23

## Class and interface abstraction

### Abstract Classes, Abstract Methods

- Abstract classes
  - Cannot be instantiated  
 new A () → ERROR if A abstract
  - Can be subclassed
- Abstract methods
  - Methods without code, they are declared but not defined
  - Must be defined in some subclass
- An abstract class can have non-abstract methods
- An abstract method must belong to an abstract class
- If a class does not implement an inherited abstract method, it must be declared abstract too

25

### Abstract Classes: Example

```
abstract class Figure {
    abstract double area ();
}

class Circle extends Figure {
    Point2D center; double radio;
    double area () {
        return Math.PI * radio * radio;
    }
}

class Triangle extends Figure {
    Point2D a, b, c;
    double area () {
        return Math.abs ((b.x-a.x)*(c.y-a.y) -
            (b.y-a.y)*(c.x-a.x))/2;
    }
}
```

26

### What are Abstract Methods Useful for?

```
class CompositeObject {
    private ArrayList figures = new ArrayList ();
    void addFigure (Figure obj) {
        figures.add (obj);
    }
    double area () {
        double a = 0;
        Iterator iter = figures.iterator ();
        while (iter.hasNext ())
            a += ((Figure) iter.next ()).area ();
        return a;
    }
}
```

27

### Inherited Abstract Methods

```
abstract class Figure {
    abstract double area ();
}

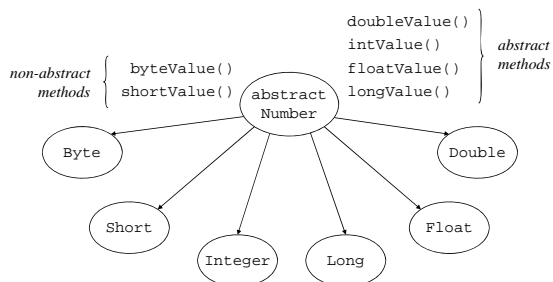
class Polygon extends Figure {
    // area () is not implemented
}

class Triangle extends Polygon {
    Point2D a, b, c;
    double area () {
        return Math.abs ((b.x-a.x)*(c.y-a.y) -
            (b.y-a.y)*(c.x-a.x))/2;
    }
}
```

Error: Polygon must be abstract

28

### java.lang.Number is an Abstract Class



29

### Interfaces

## Motivation

```
// Sorting algorithmh

public class Algorithms {
    public static void sort (double values[]) {
        int i, j;
        for (i = 0; i < values.length; i++)
            for (j = values.length-1; j > i; j--)
                if (values[j] < values[j-1])
                    swap (values, j, j-1);
    }
}
```

31

```
// Generalization

public class Algorithms {
    public static void sort (Sortable values[]) {
        int i, j;
        for (i = 0; i < values.length; i++)
            for (j = values.length-1; j > i; j--)
                if (values[j].lessThan (values[j-1]))
                    intercambiar (values, j, j-1);
    }
}
```

We need:

- A generic way to refer to sortable types
- A definition of an ordering relation for each type of sortable data

32

```
interface Sortable {
    boolean lessThan (Sortable value);
}

class Manager extends Employee implements Sortable {
    ...
    public boolean lessThan (Sortable mgr) {
        return category < ((Manager) mgr).category;
    }
}

class Figure implements Sortable {
    ...
    public boolean lessThan (Sortable fig) {
        return area () < ((Figure) fig).area ();
    }
}
```

33

```
public static void main (String args[]) {
    Manager supervisors[] = {
        new Manager (...),
        new Manager (...),
        new Manager (...);
    };
    Figure composite[] = {
        new Triangle (...),
        new Circle (...),
        new Rectangle (...);
    };
    Algorithms.sort (supervisors);
    Algorithms.sort (composite);
}
```

34

## What is an interface?

- Collection of undefined methods and constant values
- Similar to an abstract class where all methods are abstract and public, and all variables are public, static and final
- Subclass a class → implement an interface
- A class may implement several interfaces

35

## Interface Definition: Example

```
import java.util.*;

public interface Collection {
    public int MAX_SIZE = 500;
    public void insert (Object obj);
    public Object remove ();
}
```

36

4. Class Hierarchies

```
public class Stack implements Collection {
    private ArrayList elements = new ArrayList ();

    public void insert (Object obj) {
        if (elements.size () < MAX_SIZE)
            elements.add (obj);
    }

    public Object remove () {
        if (empty ()) return null;
        else {
            Object last = elements.get (elements.size ());
            elements.remove (last);
            return last;
        }
    }

    public boolean empty () {
        return elements.size () == 0;
    }
}
```

37

### Using an Interface

```
class A {
    Collection elements;
    A (Collection col) { elements = col; }
    void pickOne (Collection col) {
        elements.insert (col.remove ());
    }
}

class MainClass {
    public static void main (String [] ) {
        Stack p1 = new Stack ();
        Stack p2 = new Stack ();
        p1.insert ("Good morning");
        p2.insert ("Good night");
        p2.insert ("to everyone");

        A a = new A (p1);
        a.pickOne (p2);
    }
}
```

38

### What's an Interface Useful for?

- An interface enforces a protocol of methods to be implemented
- An interface defines a new type
  - The classes that implement the interface are compatible with this type
- Interfaces enable multiple inheritance with regard to the type hierarchy
  - An interface defines a type => a class may have multiple supertypes
- However the interface has no content (implementation) to be inherited
  - Methods have no code
  - Variables are inherited but are unchangeable (they are static and constant)
- An interface does not provide any functionality to an object, but the possibility to be the object of other objects' functionality

39

### Why an Interface and not a Class?

If Sortable was an abstract class:

- It would have to be subclassed from the top of the hierarchy
- The lessThan method would have to be defined in Person and Employee (otherwise they would be abstract classes)
- Problem if Person and Manager are defined by different programmers

*In general, as far as it is possible and it makes sense, an interface is better than an abstract class*

### Interfaces: Other Details

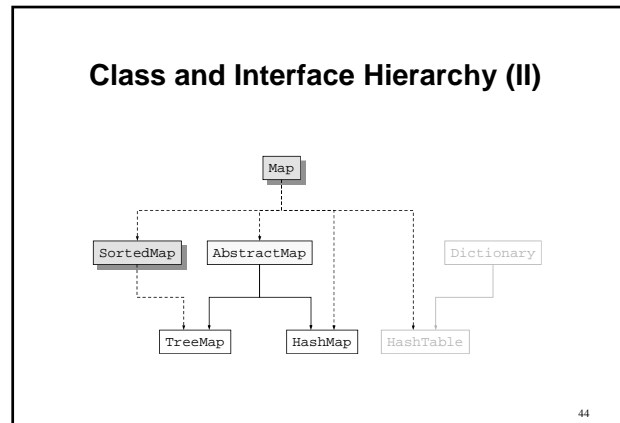
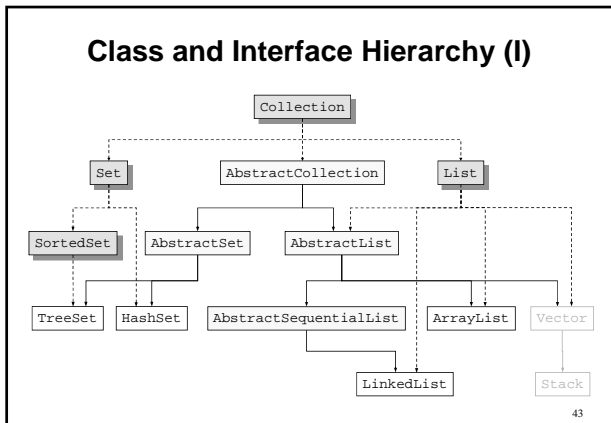
- An interface may derive from other interfaces with `extends`
- Interfaces belong to a package too
- Interfaces may be public or package
- All implemented interface methods must be declared public in the classes where they are implemented
- When implementing an interface, the definition of any coincident inherited method is lost

41

### An example of interface abstraction: The Java Collection Framework (Java 2)

- Architecture to represent and manipulate collections of data
- Interfaces: hierarchy of functionalities
- Implementations: different behaviours
- Algorithms: polymorphic, operations on the interfaces

42



### The java.util.Collection Interface

- Generic collection
- Insert, remove, membership, iteration

45

```

public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element); // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c); // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear(); // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}

public interface Iterator {
    boolean hasNext();
    Object next();
    void remove(); // Optional
}
    
```

46

### The java.util.Set Interface

- Same methods as Collection
- Elements cannot be repeated

This affects add, containsAll, addAll, retainAll, removeAll

- `set1.equals(Object set2)` may return true even if set2 has not the same implementation type as set1
- Implementations
  - HashSet: hash table, maximum efficiency
  - TreeSet: red-black tree, ensures iteration order

47

### The java.util.SortedSet Interface

- Ordered set
- `iterator()` follows an ordered iteration
- `toArray()` returns the elements in order
- `toString()` shows the elements in order
- A custom order relation (comparison function) can be specified in the constructor (see implementations)
- Implementations: TreeSet

48

## 4. Class Hierarchies

```
public interface SortedSet extends Set {
    // Range-view
    SortedSet subSet(Object fromElement, Object toElement);
    SortedSet headSet(Object toElement);
    SortedSet tailSet(Object fromElement);

    // Endpoints
    Object first();
    Object last();

    // Comparator access
    Comparator comparator();
}
```

Subsets that share memory space with the original set

49

## The java.util.List Interface

- Elements have a position in the collection
- Positional operations
- add at the end, remove the first
- list1.equals (Object list2) may return true even if list2 has not the same implementation type as list1
- Implementations
  - ArrayList: encapsulates a dynamic array (constant access, linear insertion/removal)
  - LinkedList: encapsulates a linked list (linear access, constant insertion/removal)

```
addFirst (Object obj)    addLast (Object obj)
getFirst ()              getLast ()
removeFirst ()           removeLast ()
```

50

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element); // Optional
    void add(int index, Object element); // Optional
    Object remove(int index); // Optional
    abstract boolean addAll(int index, Collection c); // Optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Backward iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

Sublist that shares memory space with the original list. If the original changes, unpredictable result

51

```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();
    void remove(); // Optional

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void set (Object obj); // Optional
    void add (Object obj); // Optional
}
```

52

## The java.util.Map Interface

- Key / value pairs
- Keys are not repeated
- map1.equals (Object map2) may return true even if map2 has not the same implementation type as map1
- Implementations
  - HashMap: hash table, maximum efficiency
  - TreeMap: red-black tree, ensures iteration order

53

```
public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();
    ...
}
```

Always returns the same Collection object. remove changes it, add does not

54

## The java.util.SortedMap Interface

- Map ordered by key
- first, last, ranges (submaps)
- iterator() follows an ordered iteration by key
- toArray() returns the elements ordered by key
- toString() shows the elements ordered by key
- Allows changing the ordering relation (comparison function)
- Implementations: TreeMap

55

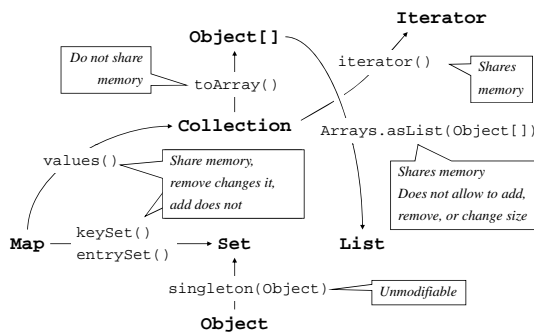
```
public interface SortedMap extends Map {
    Comparator comparator();

    Object first();           Submaps that share the memory
    Object last();           space with the original map

    // Range view
    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);
}
```

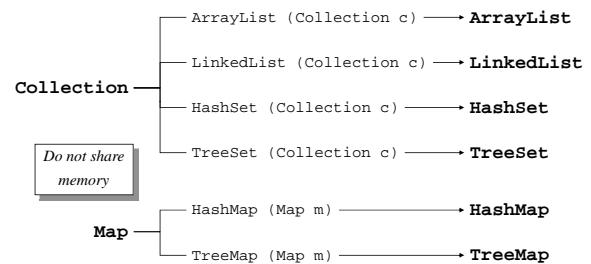
56

## Some Conversions (I)



57

## Some Conversions (II)



58

## Other views

```
// Unmodifiable
public static Collection unmodifiableCollection(Collection c);
public static Set unmodifiableSet(Set s);
public static List unmodifiableList(List list);
public static Map unmodifiableMap(Map m);
public static SortedSet unmodifiableSortedSet(SortedSet s);
public static SortedMap unmodifiableSortedMap(SortedMap m);

// Synchronized
public static Collection synchronizedCollection(Collection c);
public static Set synchronizedSet(Set s);
public static List synchronizedList(List list);
public static Map synchronizedMap(Map m);
public static SortedSet synchronizedSortedSet(SortedSet s);
public static SortedMap synchronizedSortedMap(SortedMap m);
```

59

## Collection Algorithms The java.util.Collections Interface

```
class Collections
public static void sort (List list)
public static void sort (List list, Comparator c)
public static int binarySearch (List list, Object key)
public static int binarySearch (List list, Object key,
    Comparator c)
public static Object min (Collection col)
public static Object min (Collection col, Comparator c)
public static Object max (Collection col)
public static Object max (Collection col, Comparator c)
public static List reverse (List list)
...
```

60

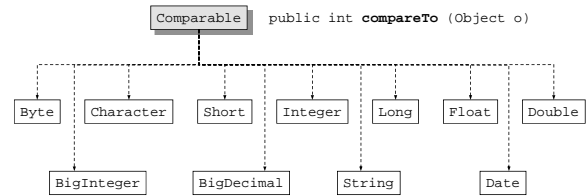
## The java.lang.Comparable Interface

```
public interface Comparable {  
    public int compareTo (Object o);  
}
```

- Endow a class with the capability to compare objects to each other (establish a natural order, define a total order in a class), for example:
  - To insert the objects in ordered collections: SortedSet or SortedMap
  - For searching and sorting algorithms
- ClassCastException if different Comparable classes are compared
- Recommended to be consistent with equals for SortedSet and SortedMap
- Some predefined Java classes implement Comparable

61

## Predefined Java Classes that implement java.lang.Comparable



62

## The java.util.Comparator Interface

```
public interface Comparator {  
    public int compare(Object o1, Object o2);  
}
```

- Getting around the natural order of a Comparable
  - In ordered collections: SortedSet or SortedMap
  - For searching and sorting algorithms
- Recommended to be consistent with equals for SortedSet and SortedMap

63

## Design of new Collection Classes

- Methods that manipulate collections
  - Input parameters: interfaces, as generic as possible
  - Return values: the most specific implementation possible
- Implementation of new types of collection
  - Choose the appropriate base abstract class
    - AbstractCollection
    - AbstractSet
    - AbstractList
    - AbstractSequentialList
    - AbstractMap
  - Implement at least all abstract methods

64

## Reflection: Class awareness at runtime

## The java.lang.Object Class

- Default parent class if none is specified
  - class A {...} is equivalent to class A extends Object {...}
- Generic object handling: array lists, etc.
- Provides default definition of methods such as equals, toString, finalize, getClass
- Arrays are subclasses of Object too

66

## Class Metainformation at Runtime

### The instanceof operator

```
Triangle t = new Triangle();
Figure fig = t;
t instanceof Triangle → true
t instanceof Polygon → true
t instanceof Figure → true
t instanceof Circle → false
fig instanceof Triangle → true
```

67

## Class Metainformation: Example

```
class A {
    void f (Person p) {
        if (p instanceof Employee)
            System.out.println ("Class Employee");
        else if (p instanceof Person)
            System.out.println ("Class Person");
    }
}
```

```
// Main block
A a = new A ();
Manager mgr = new Manager (...);
Person p = mgr;
a.f (mgr);
a.f (p);
Animal x = p;
a.f (x); // ERROR
```

68

## Class Metainformation at Runtime

### The java.lang.Class Class

- Get a Class object  
Class c1 = t.**getClass** ();  
Class c2 = fig.**getClass** ();  
Class c3 = Class.**forName** ("Triangle");  
} c1 == c2  
c1 == c3  
*method of the Object class*
- Methods of java.lang.Class  
- getName () → String  
- isArray (), isPrimitive (), isInterface ()  
- getConstructors (), getFields (),  
getMethods (), getSuperclass ()  
- newInstance ()  
*static method of the Class class*
- Class is a subclass of Object

69