List of Slides

19 The `Valuable` class
20 The `Valuable` class
21 The `Valuable` class
22 The `Valuable` class
23 Inheritance: multiple inheritance
30 The `Valuable` interface
31 The `Valuable` interface
32 Interface: definition
37 The `Valuable` interface
38 Class: is a type: and has three components
40 Interface: is a type
41 The `Valuable` interface
42 The `ValuableHouse` class
43 Interface: method implementation
44 The `ValuableHouse` class
45 The `ValuableCar` class
46 The `ValuableCar` class
47 The `Valuables` class

Java Just in Time

John Latham

February 22, 2019

Chapter 20

Interfaces, including generic interfaces

Chapter aims

- Sometimes programs appear to need **multiple inheritance**
  - would like **class** to be **subclass** of more than one **superclass**.
- Class in Java has only one superclass.
- Multiple inheritance is permitted in limited way
  - through use of **interface**s.
- We explore these here
  - including **generic interface**s.
- We also meet **generic method**s.

Section 2

Example:
Summing valuables

Aim

AIM: To introduce the idea of **multiple inheritance** and take a proper look at **interface**s. We look closely at what it means for a **class** to be a **type**, compare this with **interface**s, and revisit **method implementation**.

Summing valuables

- Outline example which requires **multiple inheritance**
  - would like **class** to be **subclass** of more than one **superclass**.
- Wish to keep track of valuables of a person, calculate total value of assets.
  - E.g. houses, cars, jewellery, artwork, etc..
- In unrelated project have **inheritance hierarchy** modelling buildings
  - including subclass `House`.
- In another unrelated project have inheritance hierarchy of vehicles
  - including subclass `Car`.
- `House` and `Car` contain much information of use in new project
  - so reuse them.

The Building class and its subclasses

```
001: // Representation of an abstract building.
002: public abstract class Building
003: {
004:
005:   // ... Lots of stuff here about buildings in general.
006:
007: } // class Building
```

```
001: // Representation of an office block.
002: public class OfficeBlock extends Building
003: {
004:
005:   // ... Lots of stuff here specific to an office block.
006:
007: } // class OfficeBlock
```

Summing valuables

```
001: // Representation of a house.
002: public class House extends Building
003: {
004:   // The number of bedrooms in the house.
005:   private int noOfBedrooms;
006:
007:
008:   // Construct a house with a given number of bedrooms.
009:   public House(int requiredNoOfBedrooms)
010:   {
011:     noOfBedrooms = requiredNoOfBedrooms;
012:   } // House
013:
014:
```

Summing valuables

```
015:   // Return the number of bedrooms in the house.
016:   public int getNoOfBedrooms()
017:   {
018:     return noOfBedrooms;
019:   } // getNoOfBedrooms
020:
021:
022:   // ... Lots more stuff here specific to a house.
023:
024: } // class House
```

The Vehicle class and its subclasses

```
001: // Representation of an abstract vehicle.
002: public abstract class Vehicle
003: {
004:
005:   // ... Lots of stuff here about vehicles in general.
006:
007: } // class Vehicle
```

```
001: // Representation of a tractor.
002: public class Tractor extends Vehicle
003: {
004:
005:   // ... Lots of stuff here specific to a tractor.
006:
007: } // class Tractor
```

Summing valuables

```
001: // Representation of a car.
002: public class Car extends Vehicle
003: {
004:   // The number of doors on the car.
005:   private final int noOfDoors;
006:
007:
008:   // Construct a car with a given number of doors.
009:   public Car(int requiredNoOfDoors)
010:   {
011:     noOfDoors = requiredNoOfDoors;
012:   } // Car
013:
014:
```

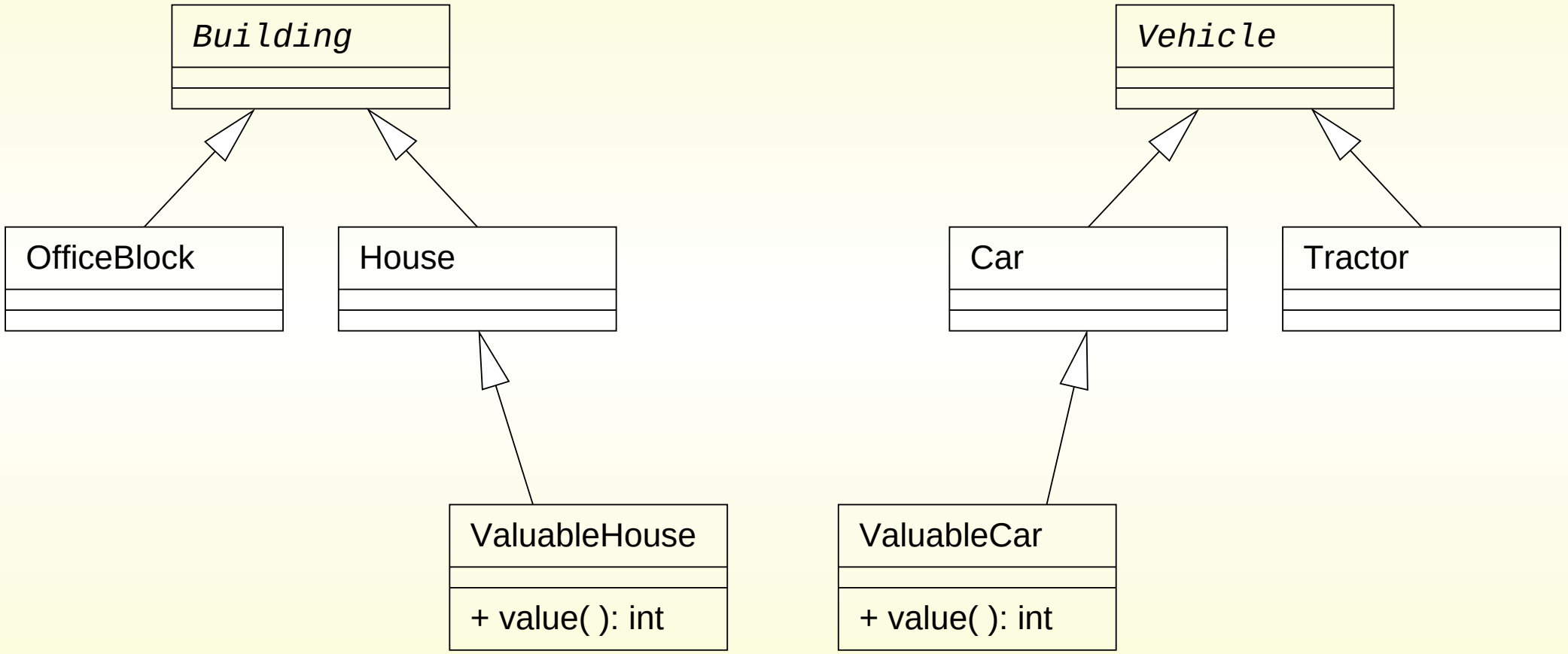
Summing valuables

```
015:   // Return the number of doors on the car.
016:   public int getNoOfDoors()
017:   {
018:     return noOfDoors;
019:   } // getNoOfDoors
020:
021:
022:   // ... Lots more stuff here specific to a car.
023:
024: } // class Car
```

The `ValuableHouse` and `ValuableCar` classes

- Other projects have lots of detail useful to calculating value of things
  - but were not actually interested in values
  - did not provide `value()` **instance method**.
- Shall add one to classes we are going to reuse
  - don't want to change existing classes
    - could interfere with previous projects.
  - Instead make new **subclass**es of `House` and `Car`
    - `ValuableHouse` and `ValuableCar`.

The ValuableHouse and ValuableCar classes



The `ValuableHouse` and `ValuableCar` classes

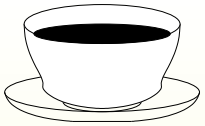
- Also have other classes for other kinds of valuables
  - `ValuableBoat`, `ValuableArtWork`, `ValuableJewellery`...
- Have capability to calculate value of house and car
  - but do not have right relationship between them
  - and other kinds of valuable items.
- To calculate total value of some valuables
  - like to have **array** of **object**s each modelling valuable item
  - each with `value()` instance method.
- The **type** of such array would have to be `Object[]`
  - `Object` is only link between `ValuableHouse` and `ValuableCar`.
  - Not every **instance** of `Object` has `value()` instance method!
- So code to add up values of items would look something like this...

ValuablesFragment.java

```
...
099: Object[] valuables;
100: // Code here to create and populate this array. ...
...
199: int total = 0;
200: for (Object someValuable : valuables)
201:   if (someValuable instanceof ValuableHouse)
202:     total += ((ValuableHouse)someValuable).value();
203:   else if (someValuable instanceof ValuableCar)
204:     total += ((ValuableCar)someValuable).value();
205:   else if (someValuable instanceof ValuableArtWork)
206:     total += ((ValuableArtWork)someValuable).value();
207:   else if // One of these for every kind of valuable, ho hum! ...
...
```

The ValuableHouse and ValuableCar classes

Coffee time: Does this surprise you? Would it be a nice idea to be able to say to the **compiler** in some simple way "trust me, `someValuable` has got a `value()` instance method, and I want to use it"? Or even more liberal, would it be nice if the compiler trusted us in the first place and just allowed us to write code to invoke the `value()` instance method of `someValuable` without moaning at us that the class `Object` does not have such an instance method?!

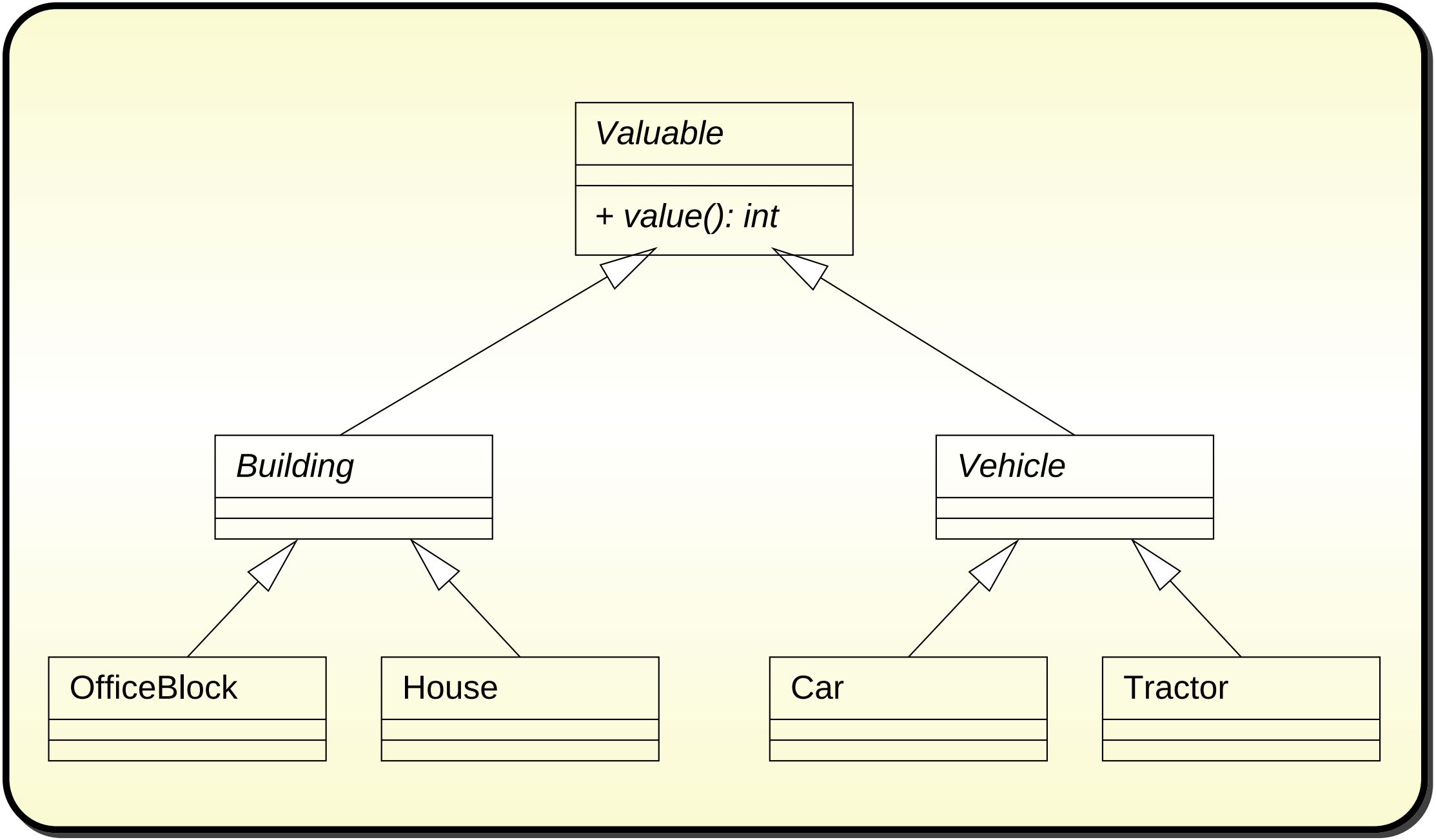


- Every time we add new kind of valuable
  - have to remember to add another bit of code in places like above
  - not acceptable position!

The `Valuable` class

- Instead want `Valuable` class
  - store `Valuable` **objects** in **array** of `Valuable[]`.
- Question: where should `Valuable` live in **inheritance hierarchy**?
- Could change approach completely
  - put `Valuable` at top of `Building` and `Vehicle`.

The Valuable class



The `Valuable` class

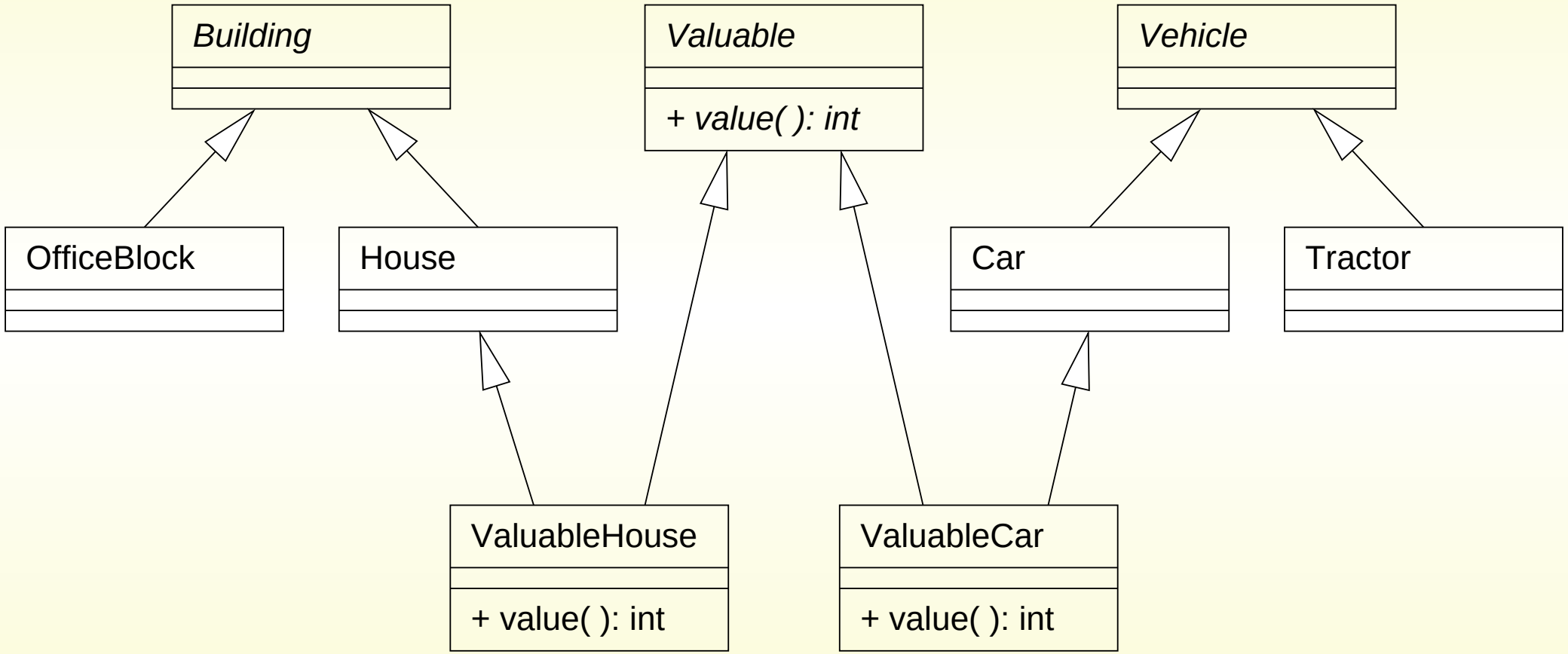
- Removed need for classes `ValuableHouse` and `ValuableCar`
  - but two bad things.
- *Second* bad thing is now have to consider meaning of value for
  - `OfficeBlock`, `Tractor`, ...
  - but only care about value for `House` and `Car`.

Coffee time: What is the *first* bad thing about this proposed inheritance hierarchy? (Hint: it would require us to do something which we have previously said we do not want to do.)

The `Valuable` class

- So, go back to idea of having classes `ValuableHouse` and `ValuableCar`.
- To get them related in most appropriate way
  - like to make them subclasses of `Valuable`
  - but not do that for other subclasses of `Building` and `Vehicle`...

The Valuable class



Inheritance: multiple inheritance

- When **class** is **subclass** of another
  - models **is a** relationship.
- Sometimes can appear natural to view class as subclass of more than one **superclass**.
  - subclass **inherit**s properties from each of its superclasses
  - known as **multiple inheritance**.

Inheritance: multiple inheritance

- But, problematic when two or more superclasses contain **instance method** with same name and **method parameter**s.
- E.g.:

```
public class Super1
{
  ...
  public void methodA()
  {
    ...
  } // methodA
  ...
} // class Super1
```

Inheritance: multiple inheritance

- And separately:

```
public class Super2
{
  ...
  public void methodA()
  {
    ...
  } // methodA
  ...
} // class Super2
```

Inheritance: multiple inheritance

- Sometime later, could make subclass of both:

```
public class Sub extends Super1, Super2
{
  ...
  public void methodB()
  {
    ...
    methodA();
    ...
  } // methodB
  ...
} // class Sub
```

Inheritance: multiple inheritance

- Two issues – first ambiguity.
- Which `methodA()` is to be called from inside `methodB()`?
- Many people regard potential for this problem as basis for view that multiple inheritance is bad idea
  - problematic **inheritance hierarchy** designs.
- Superclasses are unrelated, each has **method** with unrelated intention
  - but just happen to have same name.

Inheritance: multiple inheritance

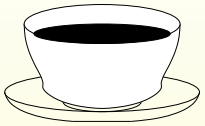
- Second issue – **run time** efficiency.
- When **virtual machine** performing **dynamic method binding**
  - needs to search inheritance hierarchy for every superclass
  - hoping there is no conflict – or dealing with any found.
  - Takes more time than searching single inheritance hierarchy.

Inheritance: multiple inheritance

- In practice, *full* multiple inheritance not very often required.
- So, Java does not permit class to have more than one superclass
  - every class, except `java.lang.Object`, has exactly one superclass
  - `Object` has none.

The `Valuable` interface

- So how implement our design?
- Java does permit *partial* multiple inheritance
  - met it in context of **graphical user interface**s.



Coffee time: You may recall that in Section **??** on page **??** we had a class `StopClock` that was both a `JFrame` and an `ActionListener`. How was that multiple inheritance achieved?

The `Valuable` interface

- Our **class** `Valuable` contains just one instance method, `value()`
  - and way we calculate value of house
    very different to way we do for car.
  - Would expect this of all subclasses of `Valuable`
  - so `value()` will be **abstract method**.
  - So `Valuable` have to be **abstract class**.
- Java has special kind of piece of code – alternative to abstract class
  - contains *only* abstract methods.

Interface: definition

- An **interface** is like **class**
  - except all **instance method**s must be **abstract method**s
  - only **method interface**s are declared.
- The **method implementation**s must be provided by each non-**abstract class** that **implement**s interface.
- E.g.. . . .

Interface: definition

```
import java.awt.event.ActionListener;
import javax.swing.JFrame;

public class StopClock extends JFrame implements ActionListener
{
  ...
  public void actionPerformed(ActionEvent event)
  {
    ...
  } // actionPerformed
  ...
} // class StopClock
```

- An **instance** of `StopClock` is **polymorphic**
  - it **is a** `StopClock`, **is a** `JFrame` and **is an** `ActionListener`.

Interface: definition

- Definition of interface has **reserved word** `interface` instead of `class`.
- Can contain list of instance method headings
  - each with no body – just semi-colon.
- Can write reserved word `abstract` in heading
  - and in instance method headings.
- But discouraged from doing so by Java language standard
  - because all instance methods *must* be abstract.
- And all instance methods *must* be **public**
  - also discouraged from writing that **modifier**.

Interface: definition

- E.g. What `ActionListener` interface might look like.

```
public interface ActionListener
{
  void actionPerformed(ActionEvent e);
} // interface ActionListener
```

Interface: definition

- Interfaces cannot contain **constructor method**s
  - nor **class method**s.
- Any **variable**s defined must be **public** `static` and **final variable**s
  - can omit those modifiers.
- Can be no **private** instance methods or variables
  - obviously?

The `Valuable` interface

Coffee time: Obviously? Why would it not make sense to have a **private** instance method or **variable** in an interface?

- Both classes and interfaces are **type**s...

Class: is a type: and has three components

- Type comprises three components
  - **set** of values
  - operations which can be performed on those values
  - **operation interface** to those operations.
- E.g. `int` is collection of numbers
  - operations such as **addition** and **multiplication**
  - **operator**s such as + and *.
- Distinction between operation and operation interface not pedantic
  - they are not same thing.
- E.g. one day Java **design**ers might permit proper multiplication symbol ($\times$) as alternative to *
  - without altering meaning of multiplication.

Class: is a type: and has three components

- Each **class** is type
  - set of all (**reference**s to) **object**s which are **instance**s of class
  - operations are **method implementation**s of **instance method**s of class
  - operation interfaces are **method interface**s.

Interface: is a type

- An **interface** is **type**
  - **set** of all (**reference**s to) **object**s
    which are **instance**s of any **class** that **implement**s the interface.
  - Operations are **method implementation**s of **instance method**s of interface
    * provided by each class which implements interface.
  - And **operation interface**s is **method interface**s defined in interface
    * (and, in effect, redefined in each class that implements it).
- An interface defines only operation interfaces of type
  - hence name – *interface*.
- Can think of as **interface contract**
  - any class that claims to implement it
    obliged to supply operation implementations.

The Valuable interface

```
001: // Objects which have a value obtained via a value() method.
002: public interface Valuable
003: {
004:   // The value of this Valuable.
005:   int value();
006:
007: } // interface Valuable
```

The ValuableHouse class

```
001: // Representation of a Valuable which is a house.
002: public class ValuableHouse extends House implements Valuable
003: {
004:   // A measure of the value of the area the house is in.
005:   private double locationDesirabilityIndex;
006:
007:
008:   // Construct a ValuableHouse with a given number of bedrooms
009:   // and location desirability.
010:   public ValuableHouse(int requiredNoOfBedrooms,
011:                        double requiredLocationDesirabilityIndex)
012:   {
013:     super(requiredNoOfBedrooms);
014:     locationDesirabilityIndex = requiredLocationDesirabilityIndex;
015:   } // ValuableHouse
```

Interface: method implementation

- Non-**abstract class** which **implement**s **interface**
  - must supply **method implementation**s for **abstract method**s in interface.
- As with making **override** of **instance method** in **superclass**
  - danger of getting **method parameter type** wrong
    - thus introducing **overloaded method** instead
  - or mistyping method name.
- The **override annotation** `@Override` extended in Java 6.0
  - enables us to tell **compiler** we believe instance method is override or implementation of one from superclass *or* interface.
- E.g. detects when have got method implementation correct
  - but forgot to say that **class** implements interface we had in mind!

The ValuableHouse class

```
018:   // Calculate and return the value of this valuable item.
019:   @Override
020:   public int value()
021:   {
022:     return (int) (getNoOfBedrooms() * 50000 * locationDesirabilityIndex);
023:   } // valuable
024:
025:
026:   // Return a short description of this as a valuable item.
027:   @Override
028:   public String toString()
029:   {
030:     return "House worth " + value();
031:   } // toString
032:
033: } // class ValuableHouse
```

The ValuableCar class

```
001: // Representation of a Valuable which is a car.
002: public class ValuableCar extends Car implements Valuable
003: {
004:   // A measure of the value of the car in general.
005:   private double streetCredibilityIndex;
006:
007:
008:   // Construct a ValuableCar with a given number of doors
009:   // and general desirability.
010:   public ValuableCar(int requiredNoOfDoors,
011:                      double requiredStreetCredibilityIndex)
012:   {
013:     super(requiredNoOfDoors);
014:     streetCredibilityIndex = requiredStreetCredibilityIndex;
015:   } // ValuableCar
```

The ValuableCar class

```
018:    // Calculate and return the value of this valuable item.
019:    @Override
020:    public int value()
021:    {
022:      return (int) (getNoOfDoors() * 2000 * streetCredibilityIndex);
023:    } // valuable
024:
025:
026:    // Return a short description of this as a valuable item.
027:    @Override
028:    public String toString()
029:    {
030:      return "Car worth " + value();
031:    } // toString
032:
033: } // class ValuableCar
```

The Valuables class

```
001: // Representation of a collection of Valuables.
002: public class Valuables
003: {
004:   // The Valuables, stored in a partially filled array, together with size.
005:   private final Valuable[] valuableArray;
006:   private int noOfValuables;
007:
008:
009:   // Create a collection with the given maximum size.
010:   public Valuables(int maxNoOfValuables)
011:   {
012:     valuableArray = new Valuable[maxNoOfValuables];
013:     noOfValuables = 0;
014:   } // Valuables
```

The Valuables class

```
017:   // Add a given Valuable to the collection (ignore if full).
018:   public void addValuable(Valuable valuable)
019:   {
020:     if (noOfValuables < valuableArray.length)
021:     {
022:       valuableArray[noOfValuables] = valuable;
023:       noOfValuables++;
024:     } // if
025:   } // addValuable
```

The Valuables class

- No **cast**ing needed – all objects definitely of type `Valuable`

```
028:   // Calculate and return the total value of the collection.
029:   public int totalValue()
030:   {
031:     int result = 0;
032:     for (int index = 0; index < noOfValuables; index++)
033:       result += valuableArray[index].value();
034:     return result;
035:   } // totalValue
```

The Valuables class

```
038:   // Return a short description of the collection.
039:   @Override
040:   public String toString()
041:   {
042:     if (noOfValuables == 0)
043:       return "Nothing valuable";
044:
045:     String result = valuableArray[0].toString();
046:     for (int index = 1; index < noOfValuables; index++)
047:       result += String.format("%n%s", valuableArray[index]);
048:     return result;
049:   } // toString
```

The Valuables class

```
052:   // Create a Valuables collection, add Valuable items and show result.
053:   // Purely for testing during development.
054:   public static void main(String[] args)
055:   {
056:     Valuables valuables = new Valuables(5);
057:
058:     // My first house -- I was so proud of its spare bedroom
059:     // and 'value for money' area.
060:     valuables.addValuable(new ValuableHouse(2, 0.5));
061:
062:     // My first car, not quite a 'head turner',
063:     // but its third door was handy when the main 2 got stuck.
064:     valuables.addValuable(new ValuableCar(3, 0.25));
065:
```

The Valuables class

```
066:     // It was nice to have a new car when I started work.
067:     valuables.addValuable(new ValuableCar(4, 1.0));
068:
069:     // Then I won the lottery! (Yeah, right.)
070:     valuables.addValuable(new ValuableHouse(6, 2.0));
071:     valuables.addValuable(new ValuableCar(12, 4.0));
072:
073:     System.out.println("My valuables are worth " + valuables.totalValue());
074:
075:     System.out.println(valuables);
076:   } // main
077:
078: } // class Valuables
```

Trying it

Console Input / Output

```
$ java Valuables
My valuables are worth 755500
House worth 50000
Car worth 1500
Car worth 8000
House worth 600000
Car worth 96000
$ _
```

Run

Section 3

Example:

Sorting a text file using an array

Aim

AIM: To introduce the idea of **total order** and the `Comparable` **interface**. We also meet the `Arrays` **class**.

Sorting a text file using an array

- Program takes input **text file**
  - produces text **sort**ed line by line to another file.
- Not write yet another implementation of sort specific to this program
  - instead generalize idea of sorting
  - use something which can sort any **array** of any sortable items!

Design: Sorting a list: total order

- A **total order** over some **data**
  - relationship between pairs of data enables it to be **sort**ed.
- Every total order, $\preceq$, has three properties:

  (Antisymmetric:) if $x \preceq y$ and $y \preceq x$, then $x = y$

  (Transitive:) if $x \preceq y$ and $y \preceq z$, then $x \preceq z$

  (Total:) $x \preceq y$ or $y \preceq x$

Design: Sorting a list: total order

- One way of modelling
  - **function** takes a pair, (x, y) yields one of three states:
    - x comes before y.
    - x and y have same placing.
    - x comes after y.
- Typically implemented in Java by **instance method** `compareTo()`
  - compares current **instance** with given other
  - yields **`int`**: negative, zero, or positive.

The `Sortable` interface?

- Could provide **type** – an **interface** –
  for all kinds of things that can be sorted.
- Each **class** that **implement**s it
  would provide own implementation for comparing pairs of that kind.
- Could look like this.. . .

Sortable.java

```
001: // A type for all things which can be sorted.
002: public interface Sortable
003: {
004:   // This method must provide a total order, and return:
005:   //        a negative int if this should be ordered before the given other,
006:   //        zero if they should have the same ordering or
007:   //        a positive int if this should be ordered after the given other.
008:   int compareTo(Sortable other);
009:
010: } // interface Sortable
```

The `SortArray` class?

- Next would write general sorting **class**
  - **sort** items in any kind of **array** of **object**s
  - as long as **implement** `Sortable` interface.
- Could look like this....
- Notice **method parameter type**: `Sortable[]`.

SortArray.java

```
001: // Provides a class method for sorting an array of any Sortable objects.
002: public class SortArray
003: {
004:   // Sort the given array from indices 0 to noOfItemsToSort - 1.
005:   public static void sort(Sortable[] anArray, int noOfItemsToSort)
006:                   throws NullPointerException, ArrayIndexOutOfBoundsException
007:   {
008:     // Each pass of the sort reduces unsortedLength by one.
009:     int unsortedLength = noOfItemsToSort;
010:     // If no change is made on a pass, the main loop can stop.
011:     boolean changedOnThisPass;
012:     do
013:     {
```

SortArray.java

```
014:       changedOnThisPass = false;
015:       for (int pairLeftIndex = 0;
016:            pairLeftIndex < unsortedLength - 1; pairLeftIndex++)
017:       {
018:         if (anArray[pairLeftIndex].compareTo(anArray[pairLeftIndex + 1]) > 0)
019:         {
020:           Sortable thatWasAtPairLeftIndex = anArray[pairLeftIndex];
021:           anArray[pairLeftIndex] = anArray[pairLeftIndex + 1];
022:           anArray[pairLeftIndex + 1] = thatWasAtPairLeftIndex;
023:           changedOnThisPass = true;
024:         } // if
025:       } // for
026:       unsortedLength--;
027:     } while (changedOnThisPass);
028:   } // sort
029:
030: } // SortArray
```

The SortArray class?

Coffee time: Any **class** could implement `Sortable`, and obviously objects of different classes are different sizes. So, how does the **compiler** know how big to make a variable of type `Sortable`? (A tricky question, or a trick one?)

The `SortArray` class?

- Idea of having interface for any types that can be sorted is so good
  - Java already has similar thing in standard **API**
  - called `Comparable` rather than `Sortable`.
- So no need for us to write own `Sortable` interface.
- Ordering provided by `compareTo()` in class that implements `Comparable`
  - known as **natural ordering** for that class.
- And **API** has beaten us to idea
  of having class method to sort array of `Comparable` items!

Standard API: `Arrays`

- `java.util.Arrays` provides various **class method**s to perform complex manipulations of **array**s.

Standard API: `Arrays: sort()`

- One **class method** in `java.util.Arrays` called `sort`
  - takes **array** of `Objects` and **sort**s them into **natural ordering**.
  - Items in array must all be **type** `Comparable`
    - and be **mutually comparable**
  - or **exception** is **throw**n. (Sadly, parameter type is `Object[]`.)
- Uses **merge sort algorithm** or **quick sort**
  - both much more efficient than **bubble sort**.
- The **class** has several more class methods called `sort`
  - one for each array of **primitive type**.
- And second version for each type, takes three **method parameter**s
  - array, and pair of **`int`** indices, $from$ and to.
  - Enables sorting of **partially filled array**s.

The `Sort` class

- Program works by
  - reading lines from input into `String` **array**
  - sort array
  - print to output file.
- Can use `Arrays.sort()` because **class** `String` **implement**s `Comparable`.
- Use **array extension** when storing lines.
- Most of program similar to previous examples
  - so leave as coursework!

Trying it

- Not thorough test: small file of examination results.

Console Input / Output

```
$ cat input.txt
Bear,Rupert     13.7%
Smith,James     51.5%
Brown,Margaret  68.2%
Jones,Stephen   87.9%
Jackson,Helen   100%
$ java Sort input.txt output.txt
$ cat output.txt
Bear,Rupert     13.7%
Brown,Margaret  68.2%
Jackson,Helen   100%
Jones,Stephen   87.9%
Smith,James     51.5%
$ _
```

Run

Coursework: Sort a text file

(Summary only)

Implement the program to **sort** a **text file**.

Section 4

Example: Translating documents

Aim

AIM: To explore **generic interfaces**, observe that `Comparable` is generic, see that `String` **implements** it, meet `equals()` from `Object` and talk about consistency with `compareTo()`. We also introduce **generic methods**, **binary search**, revisit `Arrays` and note that an **interface** can **extend** another.

Translating documents

- Program translates documents from any language to any other!
  - Okay, just changes each word for corresponding one, according to dictionary **file**.

Translating documents

Class list for Translate

Class	Description
`Translate`	The main class containing the **main method**. It makes an **instance** of `Dictionary`, from the file named as the first **command line argument**, then reads the input document from the file named as the second argument, and outputs the translated document to the file named by the third.
`DictionaryEntry`	This contains a pair of words, the first is in the source language, and the second is its translation in the target language.
`Dictionary`	This contains an **array** of `DictionaryEntry` objects, and provides an **instance method** to translate a single word.
`SearchArray`	This contains a **class method** to search any kind of `Comparable` array – it is used by `Dictionary` to find the `DictionaryEntry` corresponding to a word that needs translating.

The `DictionaryEntry` class

- `DictionaryEntry` pairs two words
  - first from source language, second is translation.
- `Dictionary` will use efficient search mechanism
  - requires `DictionaryEntry` **array** to be **sort**ed
  - so `DictionaryEntry` needs to **implement** `Comparable`.

Interface: generic interface

- A **generic interface** is **interface**
  - with **type parameter**s.
- Type parameters may be used as **type**s in declaration of **abstract method**s.
- Works in same way as **generic class**es
  - interface itself is **raw type**
  - when supply **type argument**s identify **parameterized type**.

Standard API: `Comparable` interface

- `java.lang.Comparable` provides **type** for **object**s which can be compared with similar items
  - enables general **algorithm**s to be implemented
    - e.g. **sort**ing and efficient **array** searching.
- Introduced in Java 1.2
  - Java 5.0: became **generic interface**.
- Has one **instance method** definition.

```
public interface Comparable<T>
{
  int compareTo(T o);
} // Comparable
```

Standard API: `Comparable` interface

- Any non-**abstract class** that **implement**s `java.lang.Comparable`
  - must contain `compareTo()` **method implementation**
  - providing **total order** for its objects.
- The **type parameter**, `T`: **type** of objects that can be compared
  - **class**es that (directly) implement `Comparable`
    typically supply own class name as **type argument**.
- E.g. if `SomeClass` implements `Comparable<SomeClass>`
  - means `SomeClass` provides `compareTo()`
    enabling `SomeClass` objects to compare with given other.
- If class implements `Comparable`
  - order defined by `compareTo()` known as **natural ordering**.

Standard API: `String`: implements `Comparable`

- `java.lang.String` **implement**s `java.lang.Comparable`
  - `compareTo()` provides **lexicographic order**ing
    - dictionary order, based on values of **character**s.
- Since Java 5.0 `String` implements `Comparable<String>`.

```
public final class String implements Comparable<String>
{
  ...
  @Override
  public int compareTo(String other)
  {
    ...
  } // compareTo
  ...
} // class String
```

The `DictionaryEntry` class

Coffee time: Why do you think that `String` is a **final class**?

- Generics is good!
  - **method parameter** of `compareTo()` in `String` defined to be `String`
  - **compiler** checks any argument is `String`.
- E.g., say, try to compare `String` with `Integer`
  - get **compile time error**
  - prior to Java 5.0 `compareTo()` could only test at **run time**, using **cast**!

The `DictionaryEntry` class

- Define `DictionaryEntry` to **implement** `Comparable<DictionaryEntry>`
  - `DictionaryEntry` objects can be compared with each other
  - comparison provides **total order**.
- Have to provide implementation within class.
- Also, **extend** `Pair<String, String>`!

The DictionaryEntry class

```
001: // A word from one language, paired with the equivalent one from another.
002: public class DictionaryEntry extends Pair<String, String>
003:                              implements Comparable<DictionaryEntry>
004: {
005:   // Constructor is given the words.
006:   public DictionaryEntry(String sourceLanguageWord, String targetLanguageWord)
007:   {
008:     super(sourceLanguageWord, targetLanguageWord);
009:   } // DictionaryEntry
```

The DictionaryEntry class

- The compareTo() **method implementation** based only on first word
  - every word in input searched for in Dictionary
  - search requires DictionaryEntry objects sorted by word looking for.

```
012:   // Return negative if this first word is less than other's first word,
013:   // zero if they are the same, or positive if this one is the greater.
014:   @Override
015:   public int compareTo(DictionaryEntry other)
016:   {
017:     return getFirst().compareTo(other.getFirst());
018:   } // compareTo
```

The `DictionaryEntry` class

- `compareTo()` helps efficiently find location of certain `DictionaryEntry`
  - but also have `equals()`.

Standard API: `Object: equals()`

- `java.lang.Object` contains **instance method** `equals()`
  - designed to model **equivalence** between two **object**s.

```
public boolean equals(Object other)
{
  return this == other;
} // equals
```

- Is **inherit**ed by all other classes
  - by default all **object**s have *finest* notion of equivalence
    - two objects are **equivalent** if and only if are **equal**
    - i.e. are same object.
- Often too fine
  - many classes **override** with appropriate equivalence.

Standard API: `Comparable` interface: `compareTo()` and `equals()`

- A **class** that **implement**s `java.lang.Comparable`
  - ought to have **method implementation** of `compareTo()` consistent with `equals()`.
- I.e.

  ```
  x.equals(y)
  ```

  always same as

  ```
  x.compareTo(y) == 0
  ```

The `DictionaryEntry` class

- Follow recommendation: **equivalence** consistent with `compareTo()`.
- Two `DictionaryEntry` objects **equivalent** if and only if first words equivalent
  - regardless of second words
  - deliberately circumstances for getting zero from `compareTo()`.

The DictionaryEntry class

```
021:   // Return true if and only if this and other have the same first word.
022:   // Unless other is not a DictionaryEntry,
023:   // in which case delegate to superclass.
024:   @Override
025:   public boolean equals(Object other)
026:   {
027:     if (other instanceof DictionaryEntry)
028:       return compareTo((DictionaryEntry)other) == 0;
029:     else
030:       return super.equals(other);
031:   } // equals
032:
033: } // class DictionaryEntry
```

The `DictionaryEntry` class

Coffee time: If we changed the **method parameter** of `equals()` to be of **type** `DictionaryEntry` instead of `Object`, would it still **override** the one from `Object` as we intended? Or would it instead be an **overloaded method**? What have we written that would cause a **compile time error** if we made that mistake?

The `Dictionary` class

- `Dictionary` uses **partially filled array** to store `DictionaryEntry` **object**s
  - **data** read from `BufferedReader` passed to **constructor method**.
- Some code similar to previous examples
  - but use **generic method** `Arrays.copyOf()`
    to make **new** bigger **array** when existing one full.

Method: generic methods

- A **generic method** is **method**
  - with **type parameter**s
  - written in angled brackets before **return type**.
- Similar to **generic class** type parameters
  - but apply only to method.
- When write **method call**
  - supply **type argument**s for type parameters.
- Generic methods may be defined inside generic or non-generic class.
- May be **instance method**s or **class method**s
  - of most use as **class method**s:
  - generic features of instance methods
    usually best achieved via generic **class** type parameters.

Method: generic methods

- E.g.

```
public static <T1, T2> void myGenericMethod(T1[] anArray, T2 aValue)
{
  ... Code here that uses T1 and T2 as types.
  ... Some restrictions apply,
  ... such as we cannot make instances of T1, or T2.
} // myGenericMethod
```

- So:

```
Date[] aDateArray = ...
String aString = ...

MyClassWithGenericMethod.<Date, String>myGenericMethod(aDateArray, aString);
```

- Note type arguments written *after* dot:
  - not class type parameters.

Method: generic methods

- Peculiarity – if calling method from within same class
  - have to use class name and dot (class method)
  - or **this reference** (instance method).
- But good news!
  - usually can *omit* type arguments completely
  - **compiler** can nearly always work them out. ;-)

Standard API: `Arrays: copyOf()`

- `java.util.Arrays` provides (since Java 6.0) another **class method** `copyOf`
  - makes copy of **array**.
- Is **generic method**
  - can handle any kind of **reference type** array.
- The **new** array returned can be bigger / smaller than original
  - **array element**s same as original for **array index** positions in common.

Standard API: Arrays: copyOf()

```
public static <T> T[] copyOf(T[] original, int newLength)
{
  T[] result = ... make a new array of length newLength,
               ... where result[i] = original[i]
               ...   for all 0 <= i < min(original.length, newLength)
  return result;
} // copyOf
```

- The **type parameter**, T – **type** of array elements.
- (Uses reflection to get around restrictions on use of type parameters.)
- Class also has more class methods copyOf
  - one for each array of **primitive type**.

Standard API: Arrays: copyOf()

- Useful for **array extension**:

```
SomeType[] myArray = new SomeType[INITIAL_SIZE];
...
if ... myArray is now full and I need more room
  myArray = Arrays.copyOf(myArray, myArray.length * RESIZE_FACTOR);
...
```

The Dictionary class

```
001: import java.io.BufferedReader;
002: import java.io.IOException;
003: import java.util.Arrays;
004:
005: // Reads a translation dictionary from a given BufferedReader,
006: // and provides a translateWord method.
007: public class Dictionary
008: {
009:   // We store the DictionaryEntries in a partially filled array,
010:   // and use array extension as required.
011:   // The initial size and resize factor of that array.
012:   private static final int INITIAL_ARRAY_SIZE = 50, ARRAY_RESIZE_FACTOR = 2;
013:
014:   // The array for storing the entries, and a count of the number of them.
015:   private final DictionaryEntry[] dictionaryEntries;
016:   private final int noOfDictionaryEntries;
```

The Dictionary class

```
019:   // Read lines from the given BufferedReader, split each into tab separated
020:   // pairs, create a DictionaryEntry for it and add to dictionaryEntries.
021:   public Dictionary(BufferedReader input) throws IOException, RuntimeException
022:   {
023:     DictionaryEntry[] dictionaryEntriesSoFar
024:       = new DictionaryEntry[INITIAL_ARRAY_SIZE];
025:     int noOfDictionaryEntriesSoFar = 0;
026:     String currentLine;
027:     while ((currentLine = input.readLine()) != null)
028:     {
029:       String[] lineInParts = currentLine.split("\t");
030:       DictionaryEntry dictionaryEntry
031:         = new DictionaryEntry(lineInParts[0], lineInParts[1]);
032:       if (noOfDictionaryEntriesSoFar == dictionaryEntriesSoFar.length)
033:         dictionaryEntriesSoFar
034:           = Arrays.copyOf(dictionaryEntriesSoFar,
035:                           dictionaryEntriesSoFar.length * ARRAY_RESIZE_FACTOR);
036:       dictionaryEntriesSoFar[noOfDictionaryEntriesSoFar] = dictionaryEntry;
037:       noOfDictionaryEntriesSoFar++;
038:     } // while
```

The Dictionary class

```
039:
040:     // Sort the array to allow for efficient searching of it.
041:     Arrays.sort(dictionaryEntriesSoFar, 0, noOfDictionaryEntriesSoFar);
042:     noOfDictionaryEntries = noOfDictionaryEntriesSoFar;
043:     dictionaryEntries = dictionaryEntriesSoFar;
044:   } // Dictionary
```

The Dictionary class

- The **compiler** able to figure out **type parameter** for **generic method**
  - **method call** equivalent to

```
dictionaryEntriesSoFar
  = Arrays.<DictionaryEntry>copyOf
                (dictionaryEntriesSoFar,
                 dictionaryEntriesSoFar.length * ARRAY_RESIZE_FACTOR);
```

Coffee time: Why did we use two **local variable**s in the constructor method which we copied into the **instance variable**s at the end of it – could we instead have used the instance variables directly throughout the constructor method?

The `Dictionary` class

- Translating given word
  - **array search** for matching `DictionaryEntry`
  - if found, **return** paired second word
  - else return given word with square brackets around.
- Generalise efficient **array** search to work for array of any `Comparable` **type**
  - write in separate reusable `SearchArray` class
  - **class method** `search()`, takes three **method parameter**s:
    - array, number of items in array, entry to look for.
    - Returns **array index** of object matching
    - or negative number if no such object in array.
- For efficient searching to work, array *must* be **sort**ed by **natural ordering**
  - ensured so at end of **constructor method**.

The Dictionary class

```
047:   // Translate one word.
048:   public String translateWord(String word)
049:   {
050:     int dictionaryEntryIndex
051:       = SearchArray.search(dictionaryEntries, noOfDictionaryEntries,
052:                            new DictionaryEntry(word, null));
053:     if (dictionaryEntryIndex < 0)
054:       return "[" + word + "]";
055:     else
056:       return dictionaryEntries[dictionaryEntryIndex].getSecond();
057:   } // translateWord
058:
059: } // class Dictionary
```

Design: Searching a list: binary search

- Searching for item in **list**, previously seen **linear search**
  - if items **sort**ed in known **total order** can use **binary search**
    - far more efficient
    - but more complicated.
- Two indices `low` and `high`
  - start off **index**ing first and last elements of **data**
  - item looking for always between `low` and `high`, if present.
  - Look half way between
    - may be what looking for?
    - If **less than** wanted one, move `low` up
    - otherwise move `high` down.
    - If `low` and `high` meet – item is not there.

Design: Searching a list: binary search

```
list = ... items are stored in the list in ascending order
searchItem = .. the item we wish to find in list
int lowIndex = 0
int highIndex = list.length - 1
int midIndex = (lowIndex + highIndex) / 2
while lowIndex < highIndex && list[midIndex] != searchItem
  if list[midIndex] < searchItem
    lowIndex = midIndex + 1
  else
    highIndex = midIndex - 1
  midIndex = (lowIndex + highIndex) / 2
end-while
if list[midIndex] == searchItem
  ... you found it
else
  ... searchItem is not in the list
```

The SearchArray class

- Class method intended to handle any type of `Comparable` items
  - so **generic method**
  - single **type parameter** – `ArrayType`.
  - Require that **type argument** supplied (or implied)
    - is **class** / **interface** that **implement**s / **extend**s `Comparable`.

Interface: extending another interface

- An **interface** can **extend** another
  - **abstract method**s and **class constant**s in **superinterface**
    - **inherit**ed in **subinterface**.
- For **polymorphism**:
  - (references to) **instance**s of **class** which **implement**s subinterface
    - members of superinterface **type** as well.
- Interfaces can extend *many* other interfaces.

Class: generic class: bound type parameter: extends some interface

- A **type parameter** may be declared to **extend** some known **type**
  - may be **class** or **interface**.
- Use **reserved word** `extends` even if known type is interface.
- An **interface** is **type** just as class is.
  - Type can be **extension** of another through **inheritance**
    - by being **subclass** of another class
    - **subinterface** of another interface
    - or class that **implement**s an interface.
- If known type is interface
  - **compiler** checks supplied **type argument** is
    - class which implements the interface
    - or is that interface
    - or interface that extends it.

Method: generic methods: bound type parameter

- The **type parameter**s of **generic method** can be **bound type parameter**s.
- E.g. **class method**: **return** largest element of **array**
  - of items `Comparable` with themselves...

Method: generic methods: bound type parameter

```
public class MaxArray
{
  public static <ArrayType extends Comparable<ArrayType>>
                ArrayType getMax(ArrayType[] anArray)
                throws IllegalArgumentException
  {
    try
    {
      ArrayType result = anArray[0];
      for (int index = 1; index < anArray.length; index++)
        if (result.compareTo(anArray[index]) < 0)
          result = anArray[index];
      return result;
    } // try
    catch (ArrayIndexOutOfBoundsException e)
    { throw new IllegalArgumentException("Array must be non-empty", e); }
    catch (NullPointerException e)
    { throw new IllegalArgumentException("Array must exist", e); }
  } // getMax

} // class MaxArray
```

Method: generic methods: bound type parameter

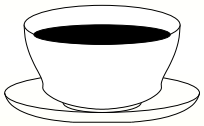
- And called:

```
String[] aStringArray = { "the", "cat", "vaporized", "on", "the", "mat" };
String maxInAStringArray = MaxArray.getMax(aStringArray);
```

- The **compiler** figured out **type argument**
  - above **method call** equivalent to

```
String maxInAStringArray = MaxArray.<String>getMax(aStringArray);
```

The SearchArray class



Coffee time: Why 'vaporized' and not 'sat'? (There is a good reason – look at the 'test data' carefully.)

The SearchArray class

```
001: // Provides an efficient search for a Comparable in a sorted Comparable[].
002: public class SearchArray
003: {
004:   // Use binary search to find searchItem in anArray which must be sorted.
005:   // Returns a negative number if not present, or array index.
006:   public static <ArrayType extends Comparable<ArrayType>>
007:         int search(ArrayType[] anArray, int noOfItems, ArrayType searchItem)
008:         throws IllegalArgumentException
009:   {
010:     if (anArray == null)
011:       throw new IllegalArgumentException("Array must exist");
012:     if (noOfItems > anArray.length)
013:       throw new IllegalArgumentException("Data length > array length: "
014:                                          + noOfItems + " " + anArray.length);
015:     if (noOfItems == 0)
016:       return -1;
017:
```

The SearchArray class

```
018:     int lowIndex = 0;
019:     int highIndex = noOfItems - 1;
020:     int midIndex = (lowIndex + highIndex) / 2;
021:     while (lowIndex < highIndex && ! anArray[midIndex].equals(searchItem))
022:     {
023:       if (anArray[midIndex].compareTo(searchItem) < 0)
024:         lowIndex = midIndex + 1;
025:       else
026:         highIndex = midIndex - 1;
027:       midIndex = (lowIndex + highIndex) / 2;
028:     } // while
029:     if (anArray[midIndex].equals(searchItem))
030:       return midIndex;
031:     else
032:       return -1;
033:   } // search
034:
035: } // SearchArray
```

The Translate class

```
001: import java.io.BufferedReader;
002: import java.io.FileReader;
003: import java.io.FileWriter;
004: import java.io.IOException;
005: import java.io.PrintWriter;
006:
007: // Program to translate a document from one language to another.
008: // Translation dictionary file is first argument.
009: // Input file is the second argument, output is the third.
010: public class Translate
011: {
012:   // The main method reads lines from the dictionary and stores them,
013:   // via the Dictionary constructor. Then it reads lines from the input file,
014:   // translates each word and writes it to the output file.
```

The Translate class

```
015:    public static void main(String[] args)
016:    {
017:      BufferedReader input = null;
018:      PrintWriter output = null;
019:      try
020:      {
021:        if (args.length != 3)
022:          throw new IllegalArgumentException
023:                      ("There must be exactly three arguments:"
024:                       + " dictfile infile outfile");
025:
026:        // The dictionary.
027:        Dictionary dictionary
028:          = new Dictionary(new BufferedReader(new FileReader(args[0])));
029:
030:        input = new BufferedReader(new FileReader(args[1]));
031:        output = new PrintWriter(new FileWriter(args[2]));
```

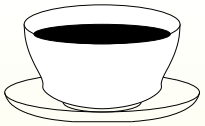
The Translate class

```
032:
033:      // Read the lines and translate each word.
034:      String currentLine;
035:      while ((currentLine = input.readLine()) != null)
036:      {
037:        String wordDelimiter = "";
038:        for (String word : currentLine.split(" "))
039:        {
040:          output.print(wordDelimiter);
041:          if (! word.equals(""))
042:            output.print(dictionary.translateWord(word));
043:          wordDelimiter = " ";
044:        } // for
045:        output.println();
046:      } // while
047:
048:    } // try
```

The Translate class

```
049:      catch (Exception exception)
050:      {
051:        System.err.println(exception);
052:      } // catch
053:      finally
054:      {
055:        try { if (input != null) input.close(); }
056:        catch (IOException exception)
057:          { System.err.println("Could not close input " + exception); }
058:        if (output != null)
059:        {
060:          output.close();
061:          if (output.checkError())
062:            System.err.println("Something went wrong with the output");
063:        } // if
064:      } // finally
065:    } // main
066:
067: } // class Translate
```

The `Translate` class



Coffee time: The **binary search** idea was a good one – it is so much faster than the **linear search** we used in earlier chapters. Do you think that the idea has already made it to the Java **API**? Check out the `Arrays` class.

Trying it

- For fun – use a `dictionary of opposites'.

Console Input / Output

```
$ cat opposites.txt
 (Output shown using multiple columns to save space.)
the     a                 boy     girl              many    no                pennies pounds
after   before            light   heavy             all     none              will    wont
dull    bright            jack    jill              and     nor               play    work
make    destroy           look    listen            themselves      others    no      yes
makes   destroys          a       many              pounds  pennies
hands   feet              you     me                work    play
$ _
```

Run

Trying it

- 'Translate' some well known cliches.

Console Input / Output

```
$ cat input.txt
all work and no play makes jack a dull boy
    while many hands make light work

if you look after the pennies
the pounds will look after themselves
$ java Translate opposites.txt input.txt output.txt
$ cat output.txt
none play nor yes work destroys jill many bright girl
    [while] no feet destroy heavy play

[if] me listen before a pounds
a pennies wont listen before others
$ _
```

Run

Coffee time: Would it be difficult to improve the program by making it able to handle capitalization and punctuation?

Coursework: Minimum and maximum `Comparable`

(Summary only)

Write a **generic method** to find the minimum and maximum items in an **array** of `Comparable` items.

Section 5

Example: Sorting valuables

Aim

AIM: To introduce the idea that a **class** can **implement** many **interface**s, and explore what it means for an **interface** to **extend** another. We also take another look at having consistency between `compareTo()` and `equals()`.

Sorting valuables

- Revisit valuables example
  - add **instance method** to `Valuables`
    sort array into descending order by value.
- The **class**es, `Building`, `Car`, `House` `OfficeBlock`, `Tractor` and `Vehicle` same as in previous version.

The ValuableHouse class?

- Can state `ValuableHouse` **implement**s `Comparable<Valuable>` as well as `Valuable`
  - so can sort with respect to other `Valuables`.

Interface: a class can implement many interfaces

- A **class** can **extend** at most one other class
  - but may **implement** any number of **interface**s
  - interfaces listed, with commas between, after **reserved word** `implements`.
- E.g. `StopClock` which automatically stops and starts when mouse moved out of / back in to window...

Interface: a class can implement many interfaces

```
import java.awt.ActionListener;
import java.awt.MouseListener;
import javax.swing.JFrame;
...
public class StopClock extends JFrame
                       implements ActionListener, MouseListener
{
  ...
  // actionPerformed is specified in the interface ActionListener
  public void actionPerformed(ActionEvent event)
  {
    ...
  } // actionPerformed

  ... Various methods here, as specified in MouseListener.

} // class StopClock
```

The ValuableHouse class?

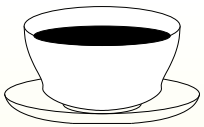
- Want `ValuableHouse` to be comparable with any other `Valuable`
  - make it implement `Comparable<Valuable>`.
  - Give definition of `compareTo()`.
  - ...

ValuableHouse.java-fragment

```
001: // Representation of a Valuable which is a house.
002: public class ValuableHouse extends House
003:                             implements Valuable, Comparable<Valuable>
004: ...
```

The ValuableHouse class?

- Dong!!!! This isn't going to work!



Coffee time: Why not?

The `Valuable` interface

- How does Java know *every* `Valuable` **implement**s `Comparable<Valuable>`?
- Make change to all `Valuable` classes
  - but that won't satisfy Java:
    * in future another class could be written that implements `Valuable` but not `Comparable<Valuable>`.
- Want to state *every* `Valuable` *must* implement `Comparable<Valuable>`
  - make `Valuable` **extend** `Comparable<Valuable>`.

The Valuable interface

```
001: // Objects which have a value obtained via a value() method.
002: public interface Valuable extends Comparable<Valuable>
003: {
004:   // The value of this Valuable.
005:   int value();
006:
007: } // interface Valuable
```

- Every **class** that implements `Valuable` must also provide **method implementation** for `compareTo()`.

The ValuableHouse class

```
001: // Representation of a Valuable which is a house.
002: public class ValuableHouse extends House implements Valuable
003: {
004:   // A measure of the value of the area the house is in.
005:   private double locationDesirabilityIndex;
006:
007:
008:   // Construct a ValuableHouse with a given number of bedrooms
009:   // and location desirability.
010:   public ValuableHouse(int requiredNoOfBedrooms,
011:                        double requiredLocationDesirabilityIndex)
012:   {
013:     super(requiredNoOfBedrooms);
014:     locationDesirabilityIndex = requiredLocationDesirabilityIndex;
015:   } // ValuableHouse
016:
017:
```

The ValuableHouse class

```
018:   // Calculate and return the value of this valuable item.
019:   @Override
020:   public int value()
021:   {
022:     return (int) (getNoOfBedrooms() * 50000 * locationDesirabilityIndex);
023:   } // valuable
024:
025:
026:   // Return a short description of this as a valuable item.
027:   @Override
028:   public String toString()
029:   {
030:     return "House worth " + value();
031:   } // toString
```

The ValuableHouse class

```
034:  // Return negative if this value is greater than other's value,
035:  // zero if they are the same, or positive if this value is the lesser.
036:  @Override
037:  public int compareTo(Valuable other)
038:  {
039:    return other.value() - value();
040:  } // compareTo
```

- Override `equals()`
  - make consistent with **method implementation** of `compareTo()`.
- Note **method parameter** of `equals()` has to be **type** `Object`.
- Two `Valuables` **equivalent** if have same value
  - regardless of kind of valuable and/or inner details
    - appropriate for *this* program.

The ValuableHouse class

```
043:   // Return true if and only if this and other have the same value.
044:   // Unless other is not a Valuable, in which case delegate to superclass.
045:   @Override
046:   public boolean equals(Object other)
047:   {
048:     if (other instanceof Valuable)
049:       return compareTo((Valuable)other) == 0;
050:     else
051:       return super.equals(other);
052:   } // equals
053:
054: } // class ValuableHouse
```

The ValuableHouse class

- That was simple way of ensuring `equals()` consistent with `compareTo()` for `Valuables`
  - two `Valuables` equivalent if have same value.
- But if compare `ValuableHouse` with `OfficeBlock`
  - will get definition of **equivalence** from (probably) `Building`.

The `ValuableCar` class

- Same modifications made to `ValuableCar`.
- Also other **class**es that **implement** `Valuable`
  - `ValuableBoat`, `ValuableArtWork`, `ValuableJewellery` etc..

The Valuables class

```
001: import java.util.Arrays;
002:
003: // Representation of a collection of Valuables.
004: public class Valuables
005: {
006:   // The Valuables, stored in a partially filled array, together with size.
007:   private final Valuable[] valuableArray;
008:   private int noOfValuables;
009:
010:
011:   // Create a collection with the given maximum size.
012:   public Valuables(int maxNoOfValuables)
013:   {
014:     valuableArray = new Valuable[maxNoOfValuables];
015:     noOfValuables = 0;
016:   } // Valuables
017:
018:
```

The Valuables class

```
019:   // Add a given Valuable to the collection (ignore if full).
020:   public void addValuable(Valuable valuable)
021:   {
022:     if (noOfValuables < valuableArray.length)
023:     {
024:       valuableArray[noOfValuables] = valuable;
025:       noOfValuables++;
026:     } // if
027:   } // addValuable
028:
029:
030:   // Calculate and return the total value of the collection.
031:   public int totalValue()
032:   {
033:     int result = 0;
034:     for (int index = 0; index < noOfValuables; index++)
035:       result += valuableArray[index].value();
036:     return result;
037:   } // totalValue
```

The Valuables class

```
038:
039:
040:   // Return a short description of the collection.
041:   @Override
042:   public String toString()
043:   {
044:     if (noOfValuables == 0)
045:       return "Nothing valuable";
046:
047:     String result = valuableArray[0].toString();
048:     for (int index = 1; index < noOfValuables; index++)
049:       result += String.format("%n%s", valuableArray[index]);
050:     return result;
051:   } // toString
```

The Valuables class

```
054:  // Sort the collection into order by value.
055:  public void sort()
056:  {
057:    Arrays.sort(valuableArray, 0, noOfValuables);
058:  } // sort
```

- Works because **array element**s of valuableArray **mutually comparable**.

The Valuables class

```
060:  // Create a Valuables collection, add Valuable items, sort, and show result.
061:  // Purely for testing during development.
062:  public static void main(String[] args)
063:  {
064:    Valuables valuables = new Valuables(5);
065:
066:    // My first house -- I was so proud of its spare bedroom
067:    // and 'value for money' area.
068:    valuables.addValuable(new ValuableHouse(2, 0.5));
069:
070:    // My first car, not quite a 'head turner',
071:    // but its third door was handy when the main 2 got stuck.
072:    valuables.addValuable(new ValuableCar(3, 0.25));
073:
074:    // It was nice to have a new car when I started work.
075:    valuables.addValuable(new ValuableCar(4, 1.0));
076:
```

The Valuables class

```
077:     // Then I won the lottery! (Yeah, right.)
078:     valuables.addValuable(new ValuableHouse(6, 2.0));
079:     valuables.addValuable(new ValuableCar(12, 4.0));
080:
081:     System.out.println("My valuables are worth " + valuables.totalValue());
082:
083:     valuables.sort();
084:
085:     System.out.println(valuables);
086:   } // main
087:
088: } // class Valuables
```

Trying it

Console Input / Output

```
$ java Valuables
My valuables are worth 755500
House worth 600000
Car worth 96000
House worth 50000
Car worth 8000
Car worth 1500
$ _
```

Run

Coursework: Analysis of `compareTo()` and `equals()`

(Summary only)

Undertake an analysis of previous uses of `compareTo()` and `equals()` **instance methods**.

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.