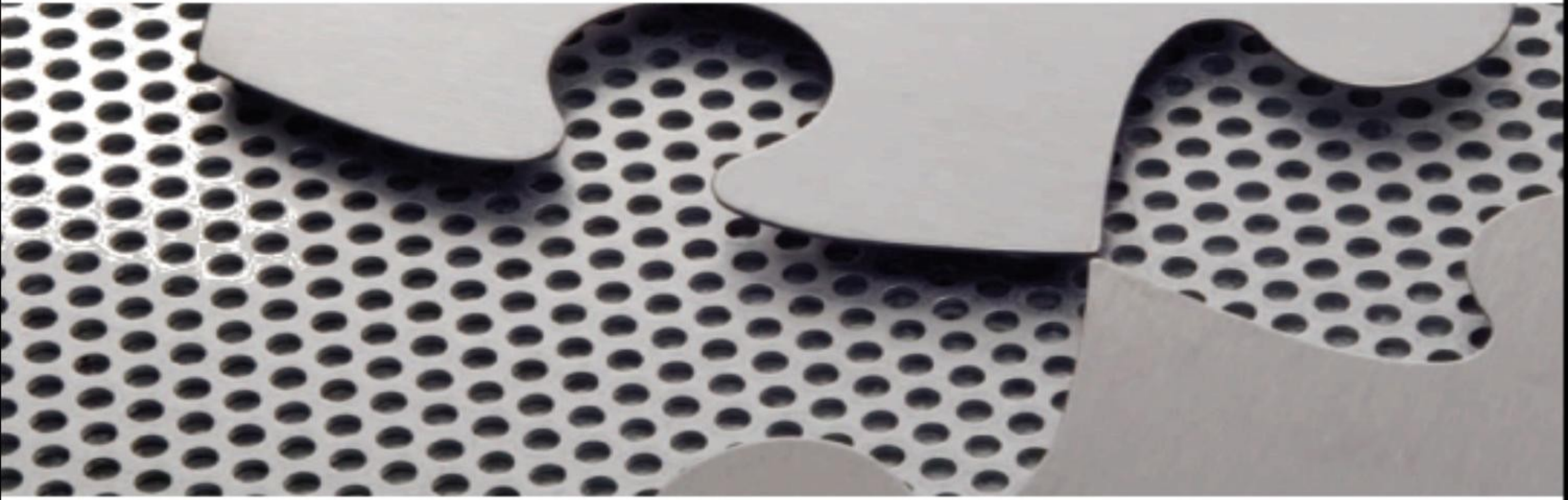
# Programming Languages Third Edition



## *Chapter 5*

## *Object-Oriented Programming*

## Objectives

- Understand the concepts of software reuse and independence
- Become familiar with the Smalltalk language
- Become familiar with the Java language
- Become familiar with the C++ language
- Understand design issues in object-oriented languages
- Understand implementation issues in object-oriented languages

## Introduction

- Object-oriented programming languages began in the 1960s with Simula
  - Goals were to incorporate the notion of an object, with properties that control its ability to react to events in predefined ways
  - Factor in the development of abstract data type mechanisms
  - Crucial to the development of the object paradigm itself

## Introduction (cont'd.)

- By the mid-1980s, interest in **object-oriented programming** exploded
  - Almost every language today has some form of structured constructs

## Software Reuse and Independence

- Object-oriented programming languages satisfy three important needs in software design:
  - Need to reuse software components as much as possible
  - Need to modify program behavior with minimal changes to existing code
  - Need to maintain the independence of different components
- Abstract data type mechanisms can increase the independence of software components by separating interfaces from implementations

## Software Reuse and Independence (cont'd.)

- Four basic ways a software component can be modified for reuse:
  - Extension of the data or operations
  - Redefinition of one or more of the operations
  - Abstraction
  - Polymorphism
- Extension of data or operations:
  - Example: adding new methods to a queue to allow elements to be removed from the rear and added to the front, to create a double-ended queue or deque

## Software Reuse and Independence (cont'd.)

- Redefinition of one or more of the operations:
  - Example: if a square is obtained from a rectangle, area or perimeter functions may be redefined to account for the reduced data needed
- Abstraction, or collection of similar operations from two different components into a new component:
  - Example: can combine a circle and rectangle into an abstract object called a figure, to contain the common features of both, such as position and movement

## Software Reuse and Independence (cont'd.)

- Polymorphism, or the extension of the type of data that operations can apply to:
  - Examples: overloading and parameterized types
- **Application framework**: a collection of related software resources (usually in object-oriented form) for developer use
  - Examples: **Microsoft Foundation Classes** in C++ and **Swing** windowing toolkit in Java

# Software Reuse and Independence (cont'd.)

- Object-oriented languages have another goal:
  - Restricting access to internal details of software components
- Mechanisms for restricting access to internal details have several names:
  - Encapsulation mechanisms
  - Information-hiding mechanisms

## Smalltalk

- Smalltalk originated from the Dynabook Project at Xerox Corp.’s Palo Alto Research Center in the early 1970s
  - Dynabook was conceived as a prototype of today’s laptop and tablet computers
- Smalltalk was influenced by Simula and Lisp
- ANSI standard was achieved in 1998
- Smalltalk has the most consistent approach to object-oriented paradigm
  - Everything is an object, including constants and the classes themselves

## Smalltalk (cont'd.)

- Can be said to be **purely** object-oriented
- Includes garbage collection and dynamic typing
- Includes a windowing system with menus and a mouse, long before this became common for PCs
- Is an interactive and dynamically oriented language
  - Classes and objects are created by interaction with the system, using a set of browser windows
  - Contains a large hierarchy of preexisting classes

## Basic Elements of Smalltalk: Classes, Objects, Messages, and Control

- Every object in Smalltalk has properties and behaviors
- **Message**: a request for service
- **Receiver**: object that receives a message
- **Method**: how Smalltalk performs a service
- **Sender**: originator of the message
  - May supply data in the form of parameters or arguments
- **Mutators**: messages that result in a change of state in the receiver object

## Basic Elements of Smalltalk (cont'd.)

- **Message passing**: process of sending and receiving messages
- **Interface**: the set of messages that an object recognizes
- **Selector**: the message name
- Syntax: object receiving the message is written first, followed by the message name and any arguments
- Example: create a new set object:

```
Set new "Returns a new set object"
```

## Basic Elements of Smalltalk (cont'd.)

- Comments are enclosed in double quotation marks
- **Show it** option: causes Smalltalk to evaluate the code you have entered
- `size` message: returns the number of elements in a set
- Can send multiple messages
  - Example: `Set new size  "Returns 0"`
  - `Set` class receives the `new` message and returns an instance of `Set`, which receives the `size` message and returns an integer

## Basic Elements of Smalltalk (cont'd.)

- **Class message**: a message sent to a class
- **Instance message**: a message sent to an instance of a class
- In prior example, `new` is a class message, while `size` is an instance message
- **Unary message**: one with no arguments
- **Keyword messages**: messages that expect arguments; name ends in a colon
  - Example:

    ```
    Set new includes: 'Hello' "Returns false"
    ```

## Basic Elements of Smalltalk (cont'd.)

- If there is more than one argument, another keyword must precede each argument
  - Keyword `at:put:` expects two arguments
- Unary messages have a higher precedence than keyword messages
  - Parentheses can be used to override precedence
- **Binary messages**: allow you to write arithmetic and comparison expressions with infix notation

## Basic Elements of Smalltalk (cont'd.)

- Examples:

```
3 + 4            "Returns 7"
3 < 4            "Returns true"
```

- Can use variables to refer to objects
- Example:

```
 | array |
array := Array new: 5.     "Create an array of 5 positions"
array at: 1 put: 10.       "Run a sequence of 5 replacements"
array at: 2 put: 20.
array at: 3 put: 30.
array at: 4 put: 40.
array at: 5 put: 50
```

## Basic Elements of Smalltalk (cont'd.)

- Temporary variables are declared between vertical bars and are not capitalized
- Statements are separated by periods
- Smalltalk variables have no assigned data type
  - Any variable can name any thing
- Assignment operator is `:=`
  - Same as Pascal and Ada

## Basic Elements of Smalltalk (cont'd.)

- A sequence of messages to the same object are separated by semicolons:

```
| array |
array := Array new: 5.    "Create an array of 5 positions"
array at: 1 put: 10;      "Run a sequence of 5 replacements, cascaded"
      at: 2 put: 20;
      at: 3 put: 30;
      at: 4 put: 40;
      at: 5 put: 50
```

- Smalltalk's variables use **reference semantics**, not **value semantics**
  - A variable refers to an object; it does not contain an object

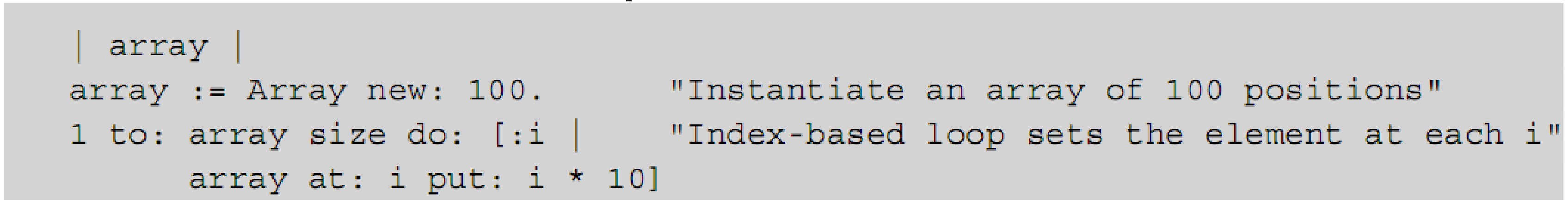
## Basic Elements of Smalltalk (cont’d.)

- Equality operator is =
- Object identity operator is ==

```
 | array alias clone |
array := #(0 0).    "Create an array containing two integers"
alias := array.     "Establish a second reference to the same object"
clone := #(0 0).    "Create a copy of the first array object"
array == alias.     "true, they refer to the exact same object"
array = alias.      "true, because they are =="
array = clone.      "true, because they refer to distinct objects with"
                       "the same properties"
array == clone         "false, because they refer to distinct objects"
clone at: 1 put: 1. "The second array object now contains 1 and 0"
array = clone          "false, because the two array objects no longer"
                       "have the same properties"
```

## Basic Elements of Smalltalk (cont'd.)

- `to:`do creates a loop



```
 | array |
array := Array new: 100.      "Instantiate an array of 100 positions"
1 to: array size do: [:i |    "Index-based loop sets the element at each i"
      array at: i put: i * 10]
```

- **Block** of code is enclosed in brackets [ ]
  - Block is similar to a lambda form in Scheme
  - Can contain arguments as **block variables**
- In Smalltalk, even control statements are expressed in terms of message passing

## Basic Elements of Smalltalk (cont'd.)

- `ifTrue:ifFalse` messages express alternative courses of action

```
 | array |
array := Array new: 100.       "Instantiate an array of 100 positions"
1 to: array size do: [:i |     "Treat odd and even positions differently"
   i odd
      ifTrue: [array at: i put: 1]
      ifFalse: [array at: i put: 2]]
```

- To print the contents of an array:

```
array do: [:element |       "Print to transcript each element followed by a return"
   Transcript nextPutAll: element printString; cr]
```

- Smalltalk includes many types of collection classes and messages for performing iterations

## The Magnitude Hierarchy

- Built-in classes are organized in a tree-like hierarchy
  - Root class is called `Object`
  - Classes descend from more general to more specific
- **Inheritance**: supports the reuse of structure and behavior
- **Polymorphism**: the use of the same names for messages requesting similar services from different classes
  - Another form of code reuse

## The Magnitude Hierarchy (cont'd.)

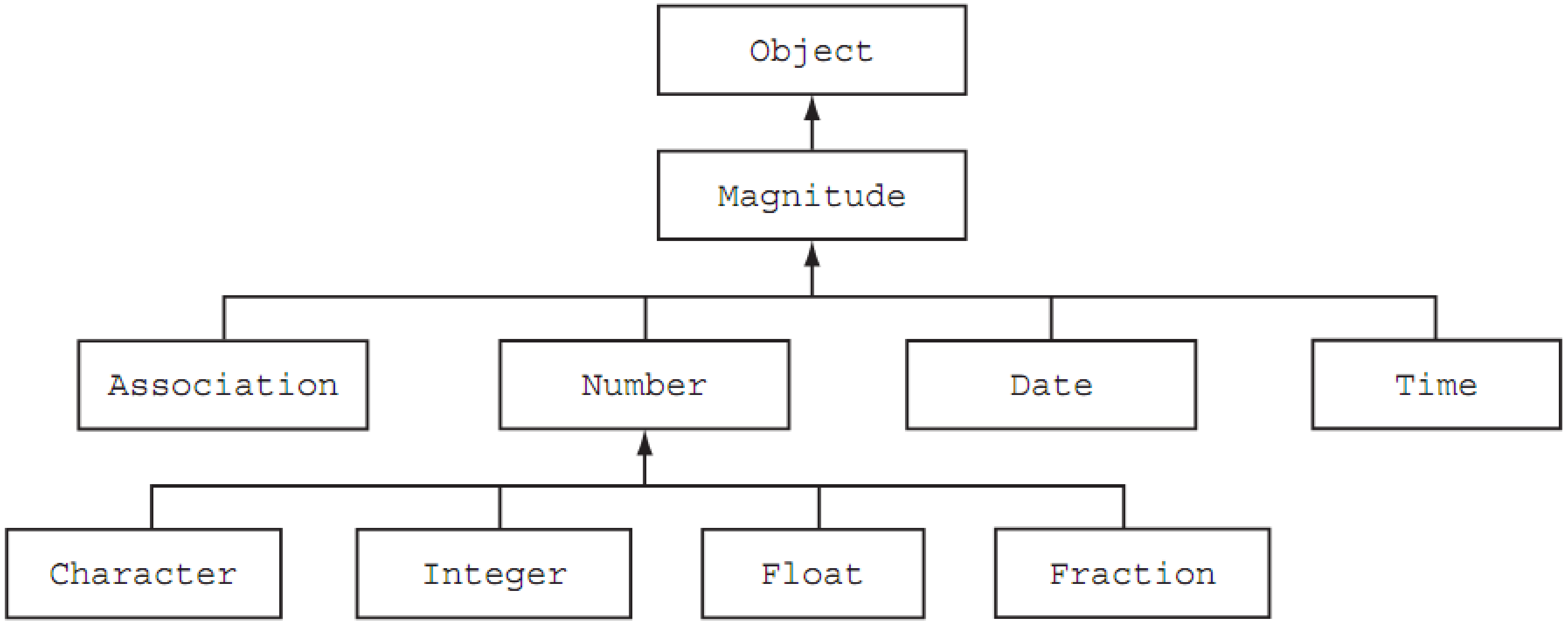
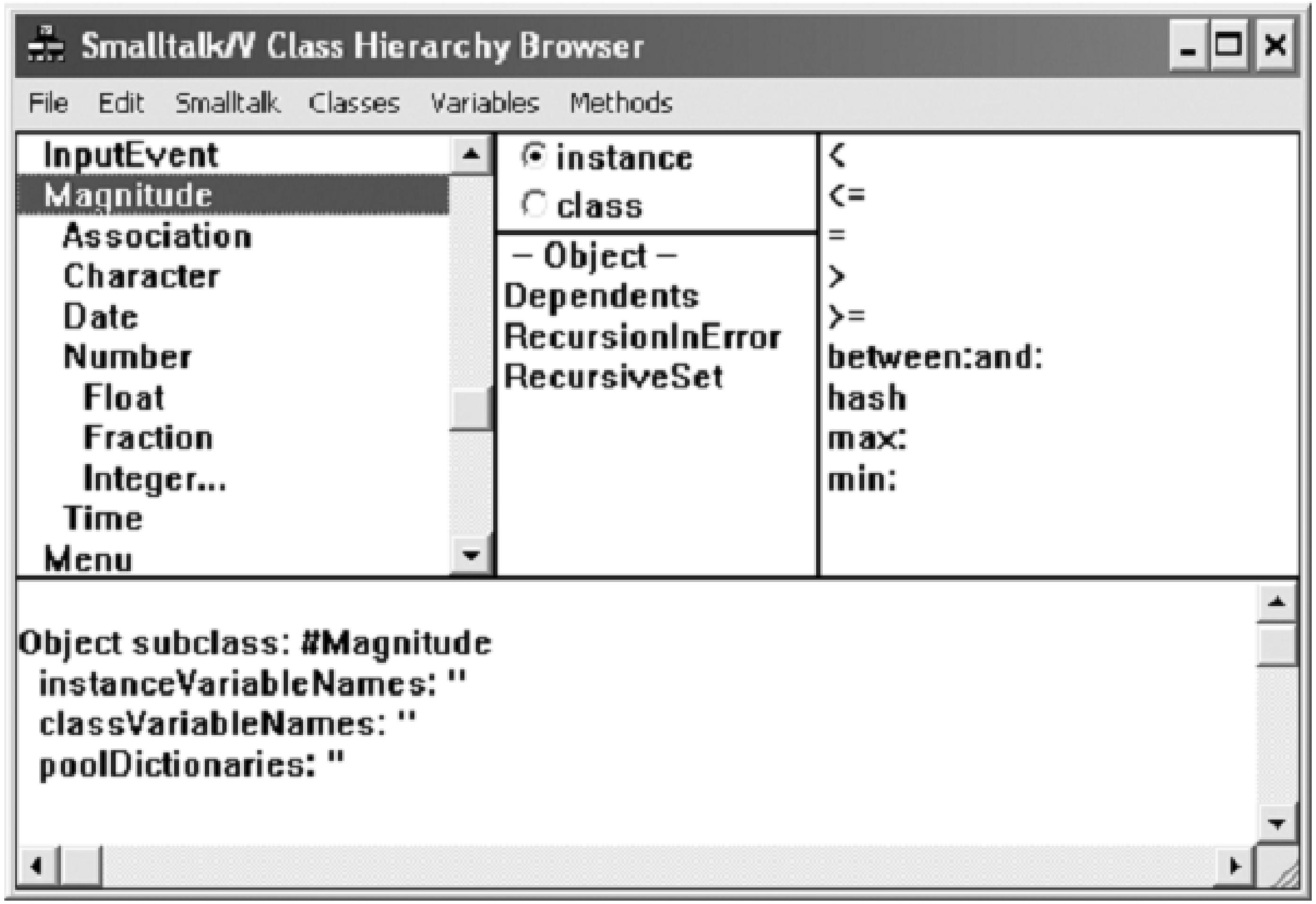


**Figure 5.1** Smalltalk's magnitude classes

## The Magnitude Hierarchy (cont'd.)

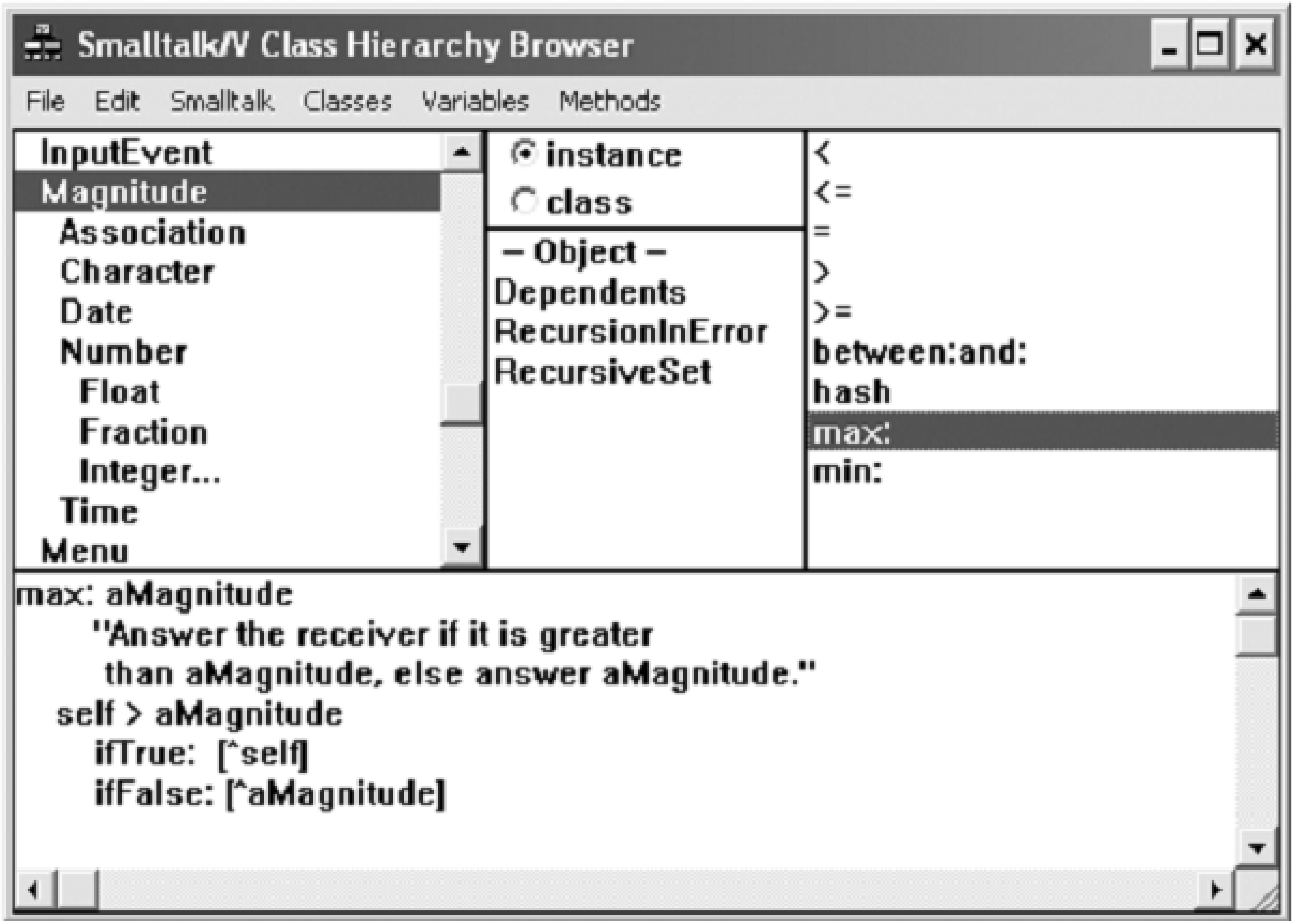
- **Concrete classes**: classes whose objects are normally created and manipulated by programs
  - Examples: `Time` and `Date`
- **Abstract classes**: serve as repositories of common properties and behaviors for classes below them in the hierarchy
  - Examples: `Magnitude` and `Number`
- Can use the Smalltalk class browser to see how classes and their methods are defined

## The Magnitude Hierarchy (cont'd.)



```
Object subclass: #Magnitude
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

**Figure 5.2** A class browser open on the magnitude classes



```
max: aMagnitude
     "Answer the receiver if it is greater
      than aMagnitude, else answer aMagnitude."
   self > aMagnitude
     ifTrue:  [^self]
     ifFalse: [^aMagnitude]
```

**Figure 5.3** The definition of the `max:` method in the `Magnitude` class

## The Magnitude Hierarchy (cont'd.)

- If we select the > operator in the message list:

```
> aMagnitude
    "Answer true if the receiver is greater
     than aMagnitude, else answer false."
  ^self implementedBySubclass
```

- When a message is sent to an object, Smalltalk binds the message name to the appropriate method
- **Dynamic** or **runtime binding**: important key to organizing code for reuse in object-oriented systems

## The Magnitude Hierarchy (cont'd.)

- To use inheritance, build a new class from an existing one
- Use **Add Subclass** option in the **Classes** menu to define a new class via inheritance

## The Collection Hierarchy

- Collections are containers whose elements are organized in a specific manner
  - Organization types include linear, sorted, hierarchical, graph, and unordered
- Built-in collections in imperative languages have historically been limited to arrays and strings
- Smalltalk provides a large set of collection types, organized in a class hierarchy
- The basic iterator is `do:`
  - It is implemented by subclasses that vary with the type of collection

# The Collection Hierarchy (cont'd.)

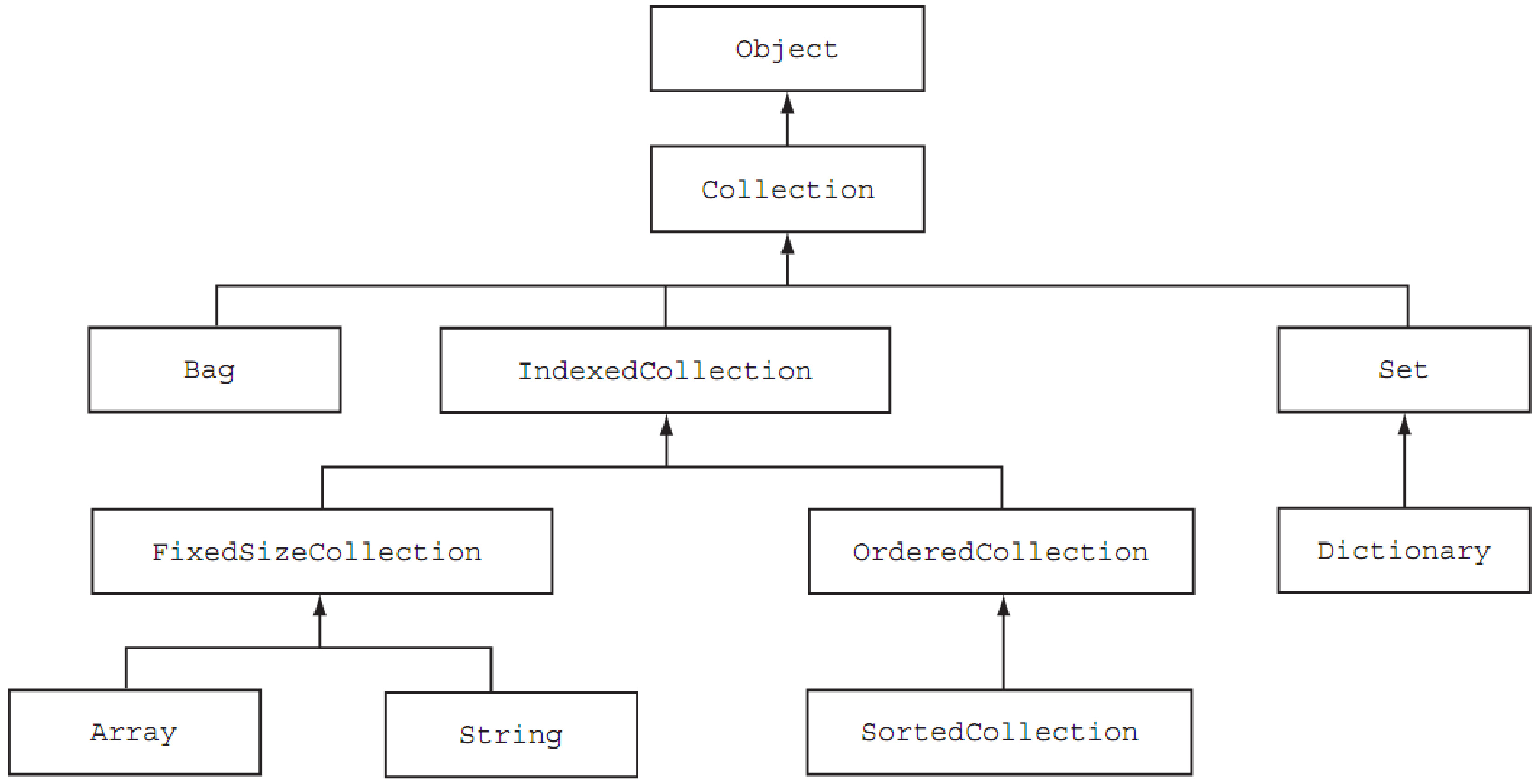


**Figure 5.4** The Smalltalk collection class hierarchy

# The Collection Hierarchy (cont'd.)

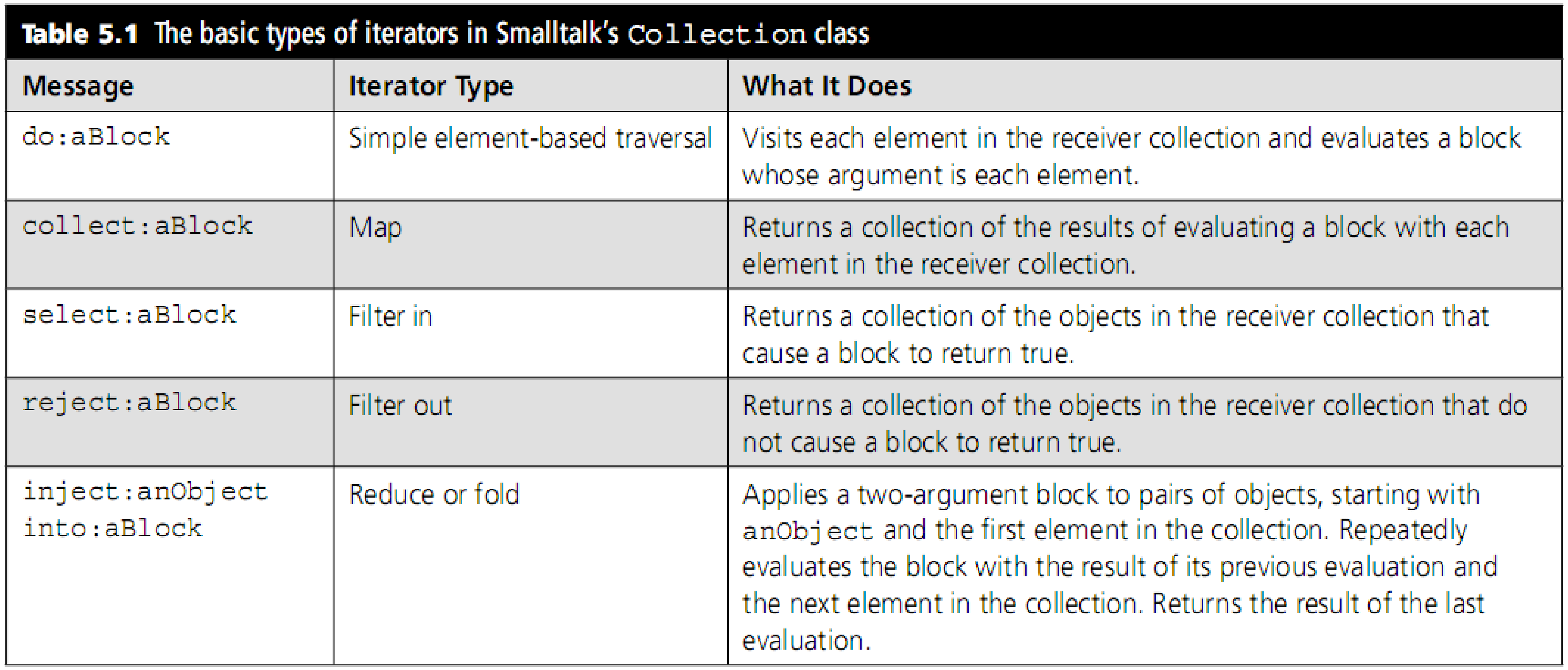


**Table 5.1** The basic types of iterators in Smalltalk's `Collection` class

| Message | Iterator Type | What It Does |
|---|---|---|
| `do:aBlock` | Simple element-based traversal | Visits each element in the receiver collection and evaluates a block whose argument is each element. |
| `collect:aBlock` | Map | Returns a collection of the results of evaluating a block with each element in the receiver collection. |
| `select:aBlock` | Filter in | Returns a collection of the objects in the receiver collection that cause a block to return true. |
| `reject:aBlock` | Filter out | Returns a collection of the objects in the receiver collection that do not cause a block to return true. |
| `inject:anObject into:aBlock` | Reduce or fold | Applies a two-argument block to pairs of objects, starting with `anObject` and the first element in the collection. Repeatedly evaluates the block with the result of its previous evaluation and the next element in the collection. Returns the result of the last evaluation. |

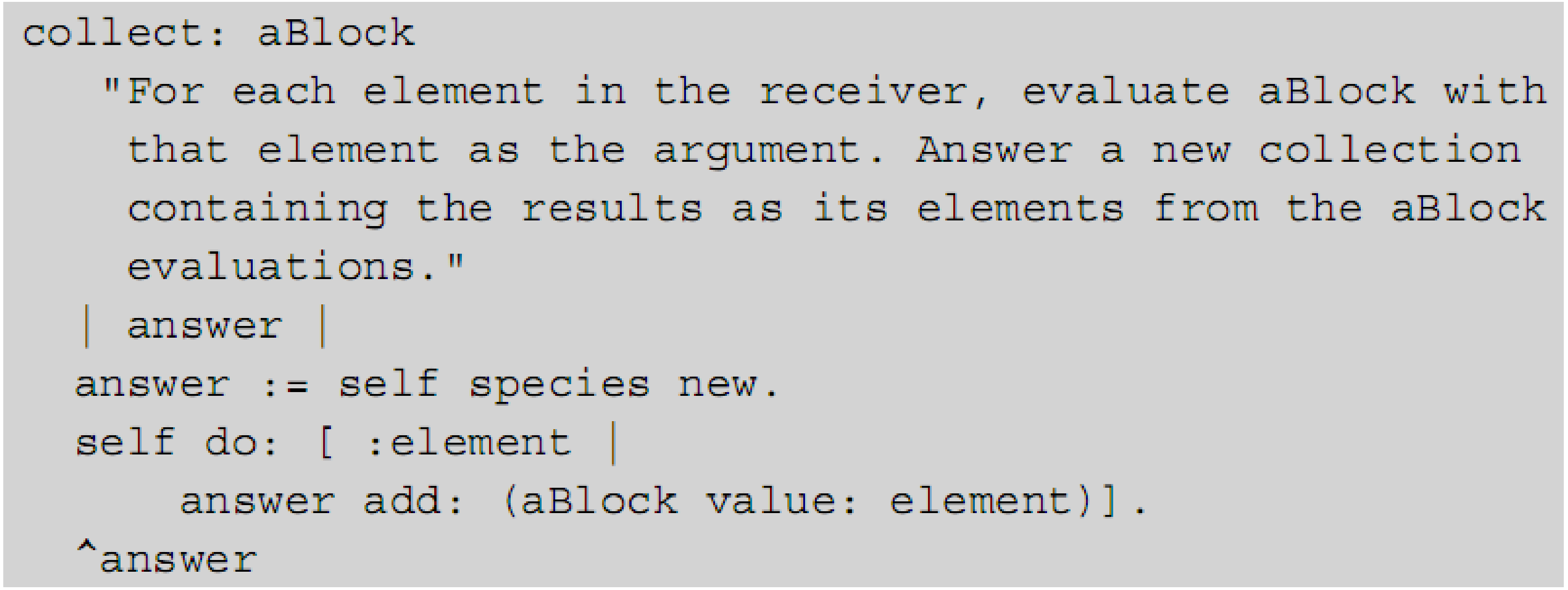
## The Collection Hierarchy (cont'd.)

- Smalltalk iterators are highly polymorphic
  - Can work with any types of collections

```
'Hi there' collect: [ :ch |
   ch asUpperCase]                              "Returns 'HI THERE'"

'Hi there' select: [ :ch |
   ch asVowel]                                  "Returns 'iee'"

'Hi there' reject: [ :ch |
   ch = Space]                                  "Returns 'Hithere'"

'Hi there' inject: Space into: [ :ch1 :ch2 |
   ch1, ch2]                                    "Returns ' H i t h e r e'"
```

# The Collection Hierarchy (cont'd.)

- Smalltalk iterators rely on the methods `do:` and `add:`
  - Example: `collect:` iterator, polymorphic equivalent of a map in functional languages



```
collect: aBlock
   "For each element in the receiver, evaluate aBlock with
    that element as the argument. Answer a new collection
    containing the results as its elements from the aBlock
    evaluations."
  | answer |
  answer := self species new.
  self do: [ :element |
      answer add: (aBlock value: element)].
  ^answer
```

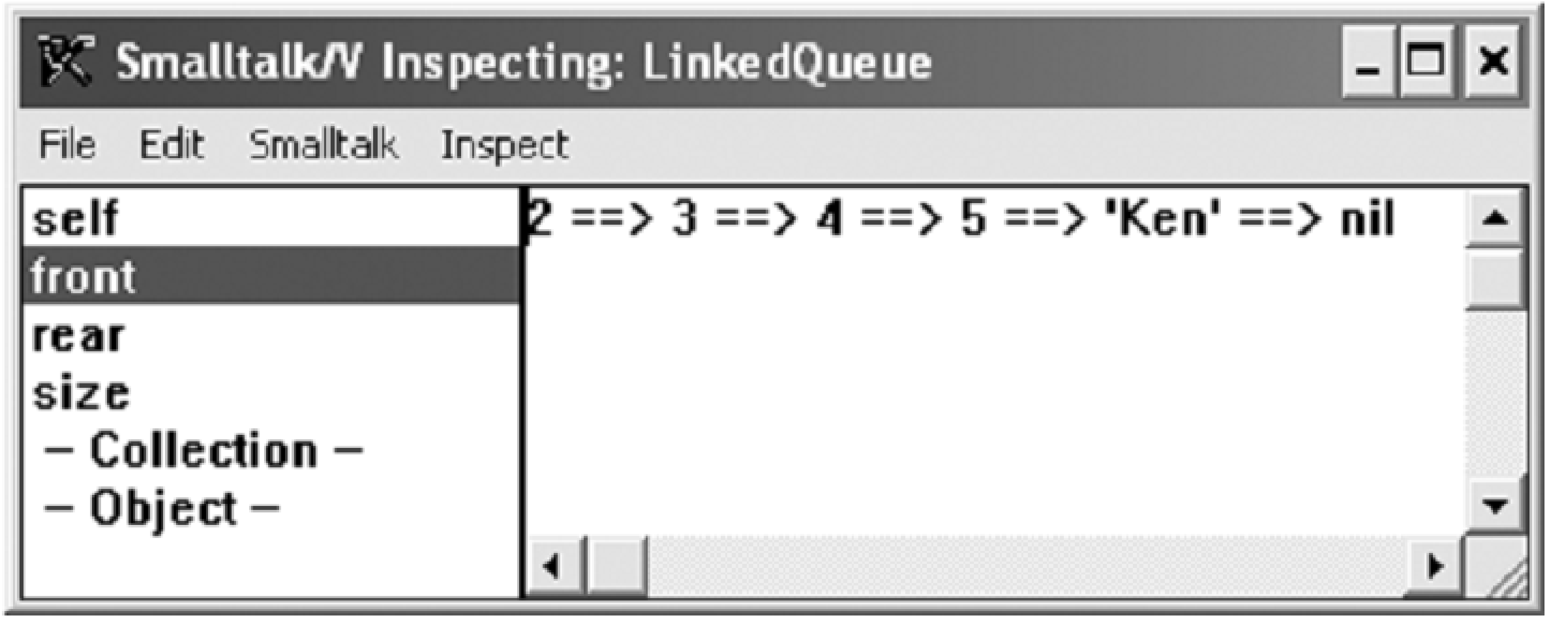
## The Collection Hierarchy (cont'd.)

- Can convert any type of collection to many of the built-in types of collections

```
asBag
   "Answer a Bag containing the elements of the receiver."
  ^(Bag new)
      addAll: self;
      yourself

addAll: aCollection
   "Answer aCollection. Add each element of
    aCollection to the elements of the receiver."
  aCollection do: [ :element | self add: element].
  ^aCollection
```

## The Collection Hierarchy (cont'd.)

- The inspect message opens an inspector window on the receiver object, to browse the values of an object's instance variables



```
2 ==> 3 ==> 4 ==> 5 ==> 'Ken' ==> nil
```

**Figure 5.5** A Smalltalk inspector window on a `LinkedQueue` object

## Java

- Originally intended as a programming language for systems embedded in appliances
  - Emphasis was on portability and small footprint: “write once, run anywhere”
- Programs compile to machine-independent byte code
- Provides conventional syntax, a large set of libraries, and support for compile-time type checking not available in Smalltalk
- Is purely object-oriented, with one exception:
  - Scalar data types (**primitive types**) are not objects

## Basic Elements of Java: Classes, Objects, and Methods

- A Java program instantiates classes and calls methods to make objects do things
- Many classes are available in standard packages
  - Programmer-defined classes can be placed in their own packages for import
- Variable definition is similar to that in C:

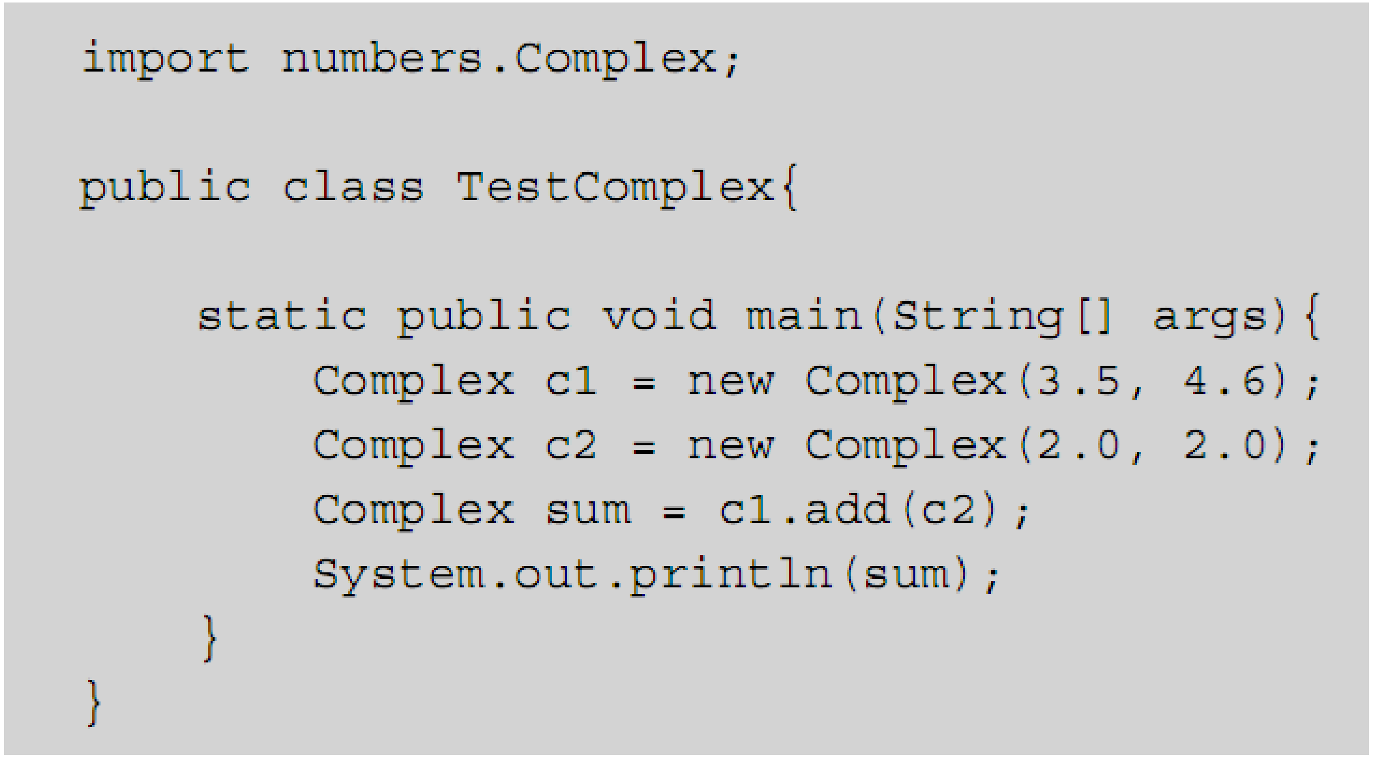
```
<class name> <variable name> =
    new <class name>(<argument-1, . ., argument-n>);
```

- Method call is similar to Smalltalk:

```
<object>.<method name>(<argument-1, . ., argument-n>)
```

## Basic Elements of Java (cont'd.)

- Example: program uses an imported class



```
import numbers.Complex;

public class TestComplex{

    static public void main(String[] args){
        Complex c1 = new Complex(3.5, 4.6);
        Complex c2 = new Complex(2.0, 2.0);
        Complex sum = c1.add(c2);
        System.out.println(sum);
    }
}
```

## Basic Elements of Java (cont'd.)

- **Class method**: a static method
- Java Virtual Machine runs the program as `TextComplex.main(<array of strings>)`
  - Command-line arguments present at launch are placed into the `args` array
- All classes inherit from the `Object` class by default
- Data encapsulation is enforced by declaring instance variables with `private` access
  - They are visible only within the class definition
- **Accessor methods**: allow programs to view but not modify the internal state of a class

## Basic Elements of Java (cont'd.)

- **Constructors**: like methods, they specify initial values for instance variables and perform other initialization actions
  - **Default constructor**: has no parameters
  - **Constructor chaining**: when one constructor calls another
- Use of `private` access to instance variables and accessor methods allows us to change data representation without disturbing other code
- Java uses reference semantics
  - Classes are also called **reference types**

## Basic Elements of Java (cont'd.)

- == is the equality operator for primitive types, and also means object identity for reference types
  - `Object` class contains an equals method that can be overridden in subclasses to implement a comparison of two distinct objects
- Methods are invoked after instantiation using dot notation:

```
z = z.add(w);
```

- Can nest operations:

```
z = z.add(w).multiply(z);
```

- Java does not allow operator overloading like C++
- Java does not allow **multimethods**, in which more than one object can be the target of a method call

## The Java Collection Framework: Interfaces, Inheritance, and Polymorphism

- **Framework**: a collection of related classes
- `java.util` package: contains the Java collection framework
- **Interface**: a set of operations on objects of a given type
  - Serves as the glue that connects components in systems
  - Contains only type name and a set of public method headers; implementer must include the code to perform the operations

# The Java Collection Framework (cont'd.)

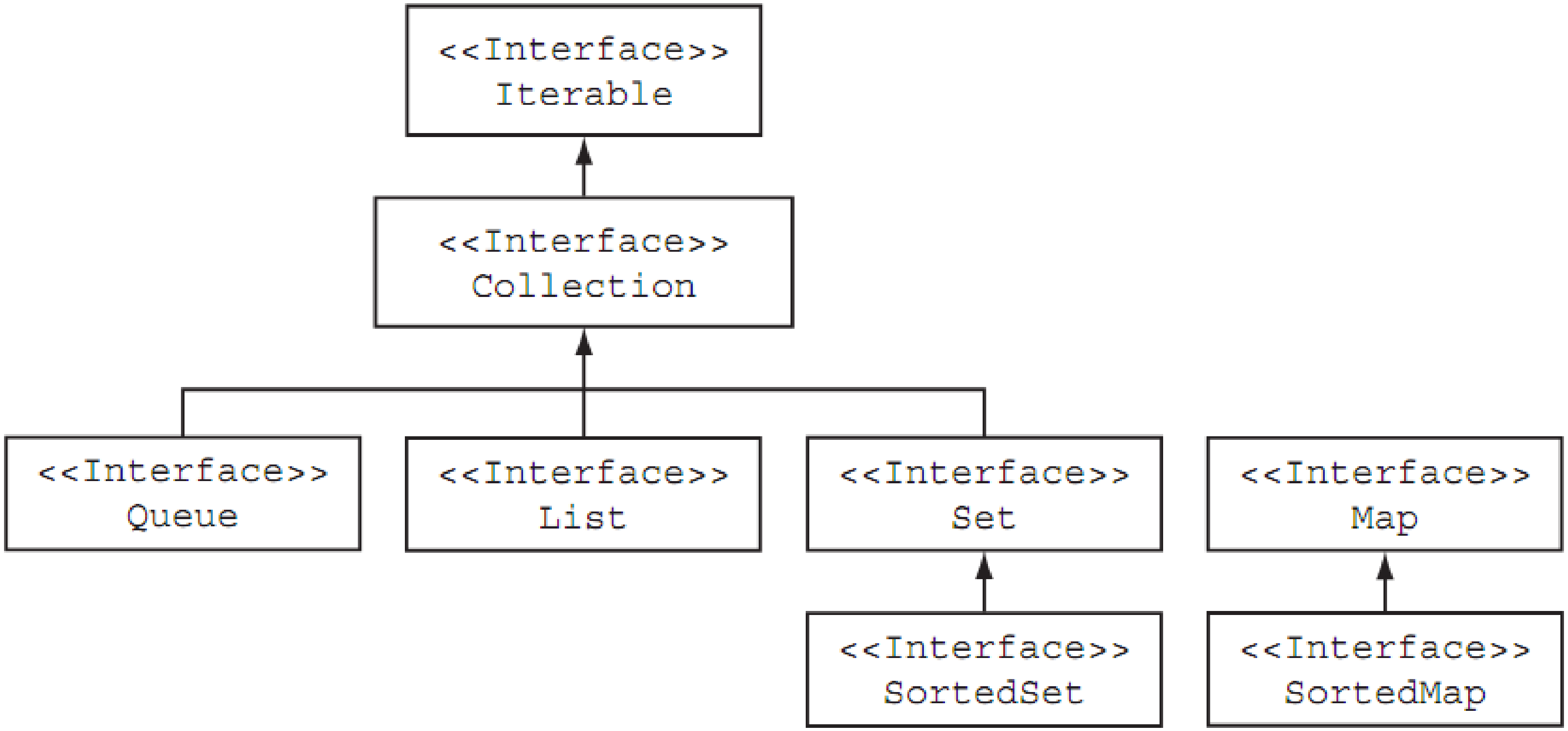
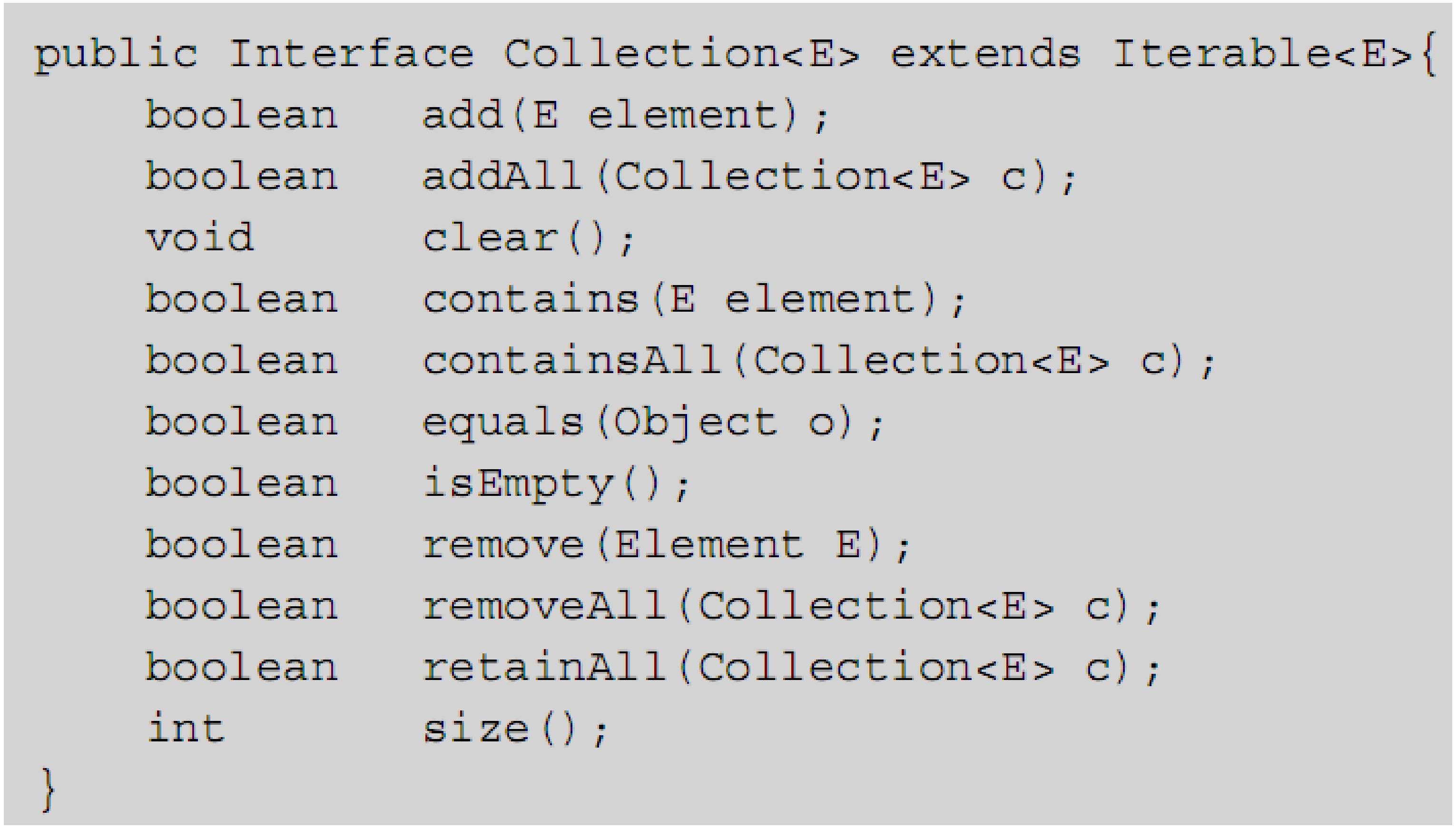


**Figure 5.6** Some Java collection interfaces

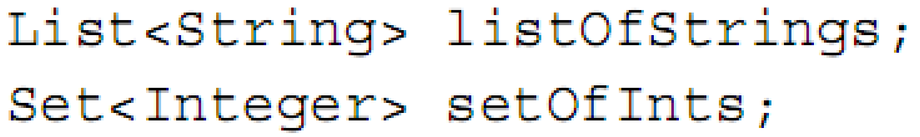
## The Java Collection Framework (cont'd.)



```
public Interface Collection<E> extends Iterable<E>{
    boolean   add(E element);
    boolean   addAll(Collection<E> c);
    void      clear();
    boolean   contains(E element);
    boolean   containsAll(Collection<E> c);
    boolean   equals(Object o);
    boolean   isEmpty();
    boolean   remove(Element E);
    boolean   removeAll(Collection<E> c);
    boolean   retainAll(Collection<E> c);
    int       size();
}
```

# The Java Collection Framework (cont'd.)

- <E> and E in the interface and method headers are **type parameters**
- Java is statically typed: all data types must be explicitly specified at compile time
- **Generic collections**: exploit parametric polymorphism
  - – **Raw collections** in early versions of Java did not
- Examples:



```
List<String> listOfStrings;
Set<Integer> setOfInts;
```

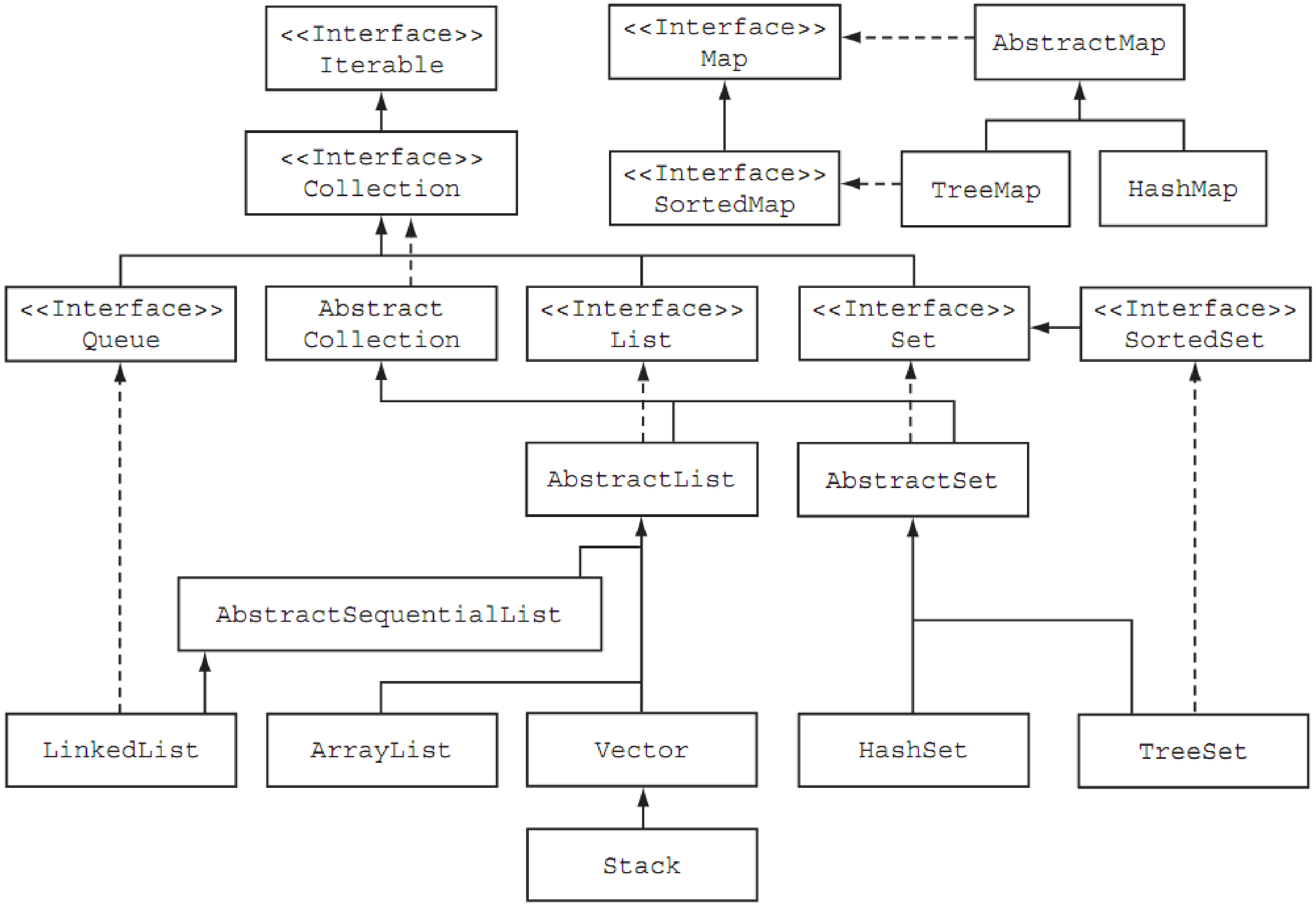


**Figure 5.7** Some Java collection classes and their interfaces

# The Java Collection Framework (cont'd.)

- Some classes, such as `LinkedList`, implement more than one interface
  - A linked list can behave as a list or as a queue

```
List<String>    listOfStrings = new LinkedList<String>();
Queue<Float>    queueOfFloats = new LinkedList<Float>();
List<Character> listOfChars   = new ArrayList<Character>();
```

- Same methods can be called on the two `List` variables even though they have different implementations and different element types
- Can only call `Queue` interface methods on the `queueOfFloats` variable because it is type `Queue`

## The Java Collection Framework (cont'd.)

- Example: develop a new type of stack class called `LinkedStack`, as a subclass of the `AbstractCollection` class
  - Gives us a great deal of additional behavior for free
- **Private inner class**: a class defined within another class
  - No classes outside of the containing class need to use it
- Java `Iterable` interface only contains the `iterator` method

## The Java Collection Framework (cont'd.)

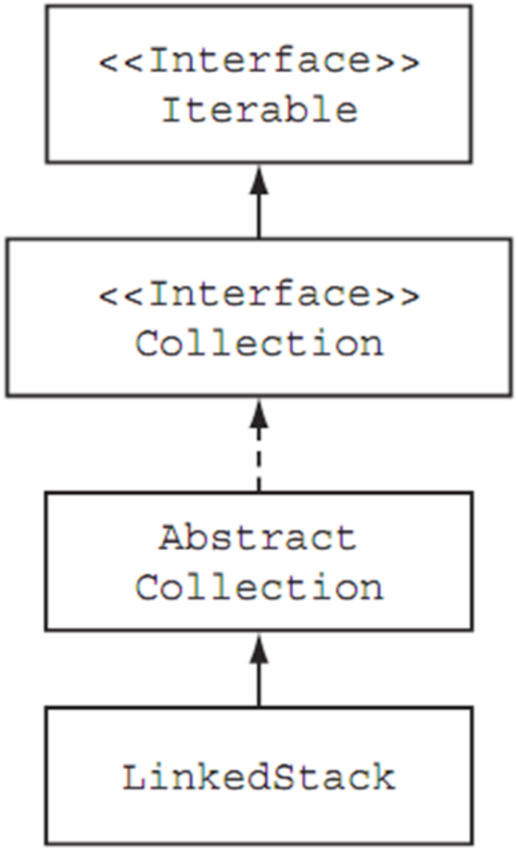


**Figure 5.8** Adding LinkedStack to the collection framework

# The Java Collection Framework (cont'd.)

- **Backing store**: the collection object on which the iterator object is operating
- Example:

```
LinkedStack<String> stack = new LinkedStack<String>();
Iterator<String> iter = stack.iterator();
```

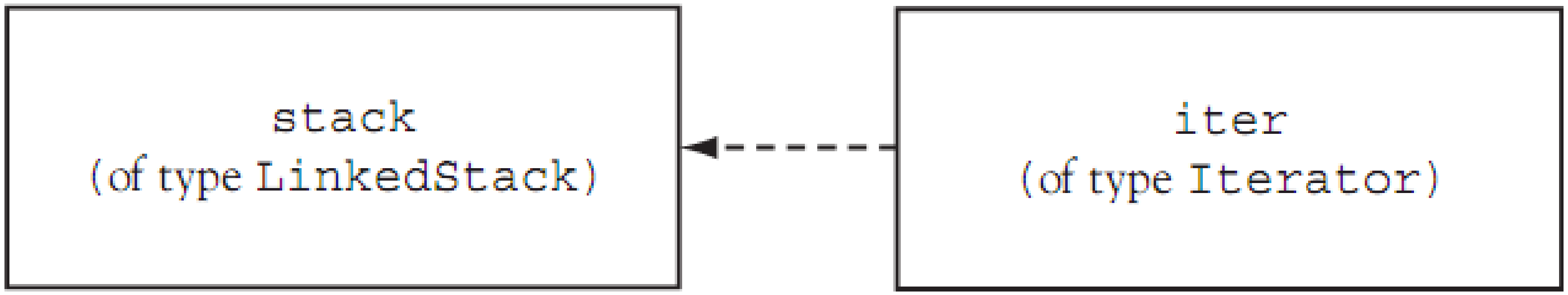


**Figure 5.9** An iterator opened on a stack object

## The Java Collection Framework (cont'd.)

- `Iterator` type is parameterized for the same element type as its backing store
  - Is an interface that specifies three methods

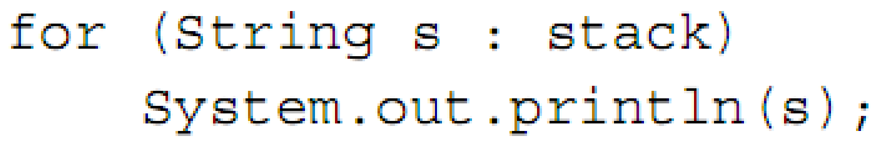
```
public Interface Iterator<E>{

    public boolean hasNext();  // True if more elements, false otherwise
    public E next();           // Returns the current element and advances
    public void remove();      // Removes the current element from the store
}
```

- To visit each element in the backing store, use `hasNext` and `next` methods

```
while (iter.hasNext())
    System.out.println(iter.next());
```

# The Java Collection Framework (cont'd.)

- Java's enhanced `for` loop is syntactic sugar for the iterator-based while loop
  - Only prerequisites to use this: the collection class must implement the `Iterable` interface and implement an `iterator` method
- Example:



```
for (String s : stack)
    System.out.println(s);
```

## Dynamic Binding and Static Binding

- **Static binding**: process of determining at compile time which implementation of a method to use by determining the object’s actual class
  - Actual code is not generated by the compiler unless the method is declared as final or static
- When Java cannot determine the object’s method at compile time, dynamic binding is used
- Java uses a jump table, which is more efficient than a search of an inheritance tree to perform dynamic binding

# Defining Map, Filter, and Reduce in Java

- Map, filter, and reduce are higher-order functions in a functional language
  - Built-in collection methods in Smalltalk
- It is possible to define `map`, `filter`, and `reduce` methods in Java using its basic iterator
  - Are defined as static methods in a special class named `Iterators`
- `map` and `filter` methods expect an input collection and return an output collection as a value
  - The actual object returned will be of the same concrete class as the input collection

# Defining Map, Filter, and Reduce in Java (cont'd.)

- How do you represent the operation that is passed as the remaining argument to these methods?
- In Java, we can define the operation as a special type of object that recognizes a method that will be called within the higher-order method's implementation

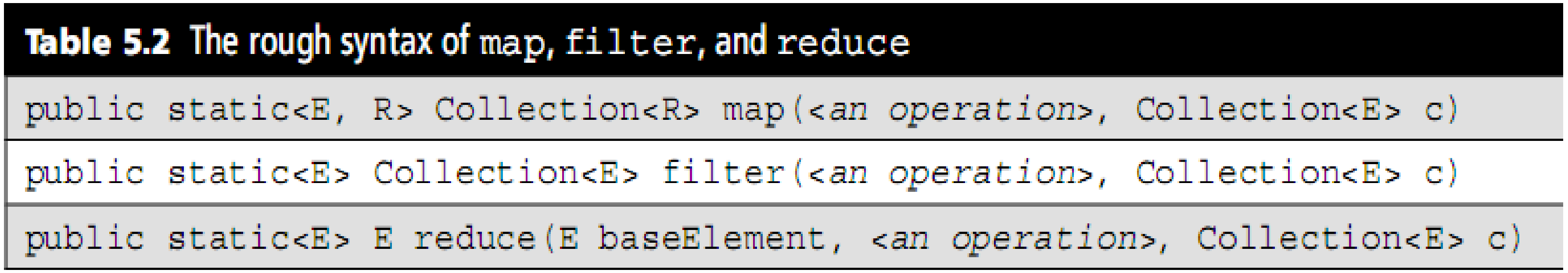


**Table 5.2** The rough syntax of `map`, `filter`, and `reduce`

| |
|---|
| `public static<E, R> Collection<R> map(`*`<an operation>`*`, Collection<E> c)` |
| `public static<E> Collection<E> filter(`*`<an operation>`*`, Collection<E> c)` |
| `public static<E> E reduce(E baseElement, `*`<an operation>`*`, Collection<E> c)` |

## Defining Map, Filter, and Reduce in Java (cont'd.)

- Three operations are needed:
  - In map, a method of one argument that transforms a collection element into another value (perhaps of a different type)
  - In filter, a method of one argument that returns a Boolean value
  - In reduce, a method of two arguments that returns an object of the same type
- Example of next slide

```
package iterators;

public interface MapStrategy<E, R>{
    // Transforms an element of type E into a result of type R
    public R transform(E element);
}

package iterators;

public interface FilterStrategy<E>{
    // Returns true if the element is accepted or false otherwise.
    public boolean accept(E element);
}

package iterators;

public interface ReduceStrategy<E>{
    // Combines two elements of type E into a result of type E.
    public E combine(E first, E second);
}
```

## Defining Map, Filter, and Reduce in Java (cont'd.)

- These strategy interfaces tell the implementer to expect an object that recognizes the appropriate method
  - Tell the user that he must only supply an object of a class that implements one of these interfaces

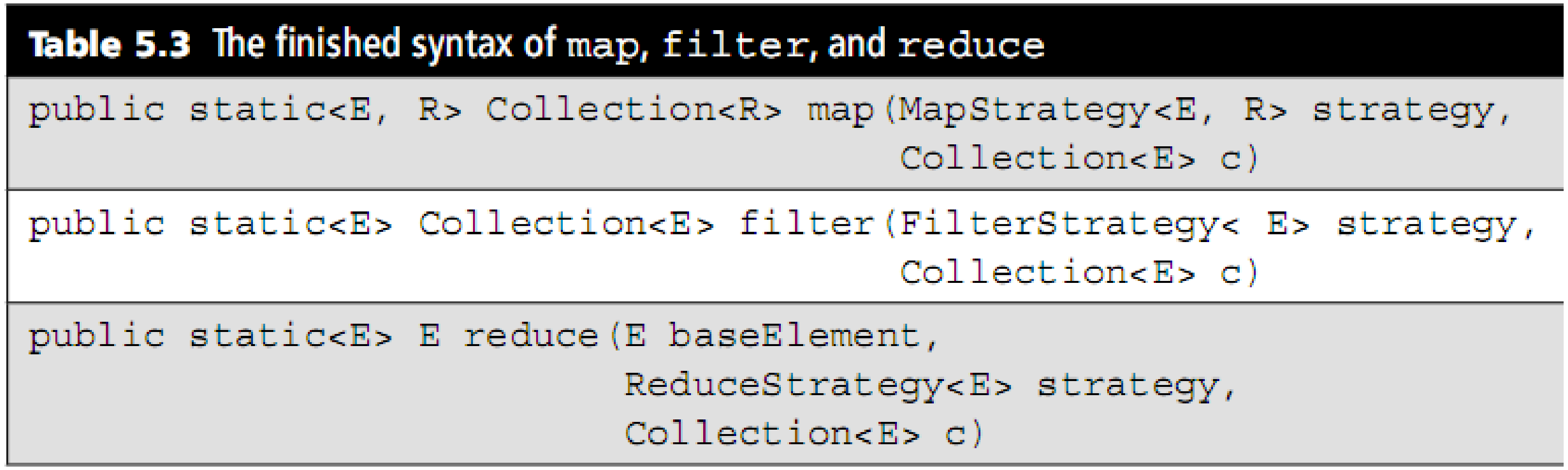


**Table 5.3** The finished syntax of `map`, `filter`, and `reduce`

| |
|---|
| `public static<E, R> Collection<R> map(MapStrategy<E, R> strategy, Collection<E> c)` |
| `public static<E> Collection<E> filter(FilterStrategy< E> strategy, Collection<E> c)` |
| `public static<E> E reduce(E baseElement, ReduceStrategy<E> strategy, Collection<E> c)` |

## Defining Map, Filter, and Reduce in Java (cont'd.)

- The `Map Strategy` interface and an example instantiation:

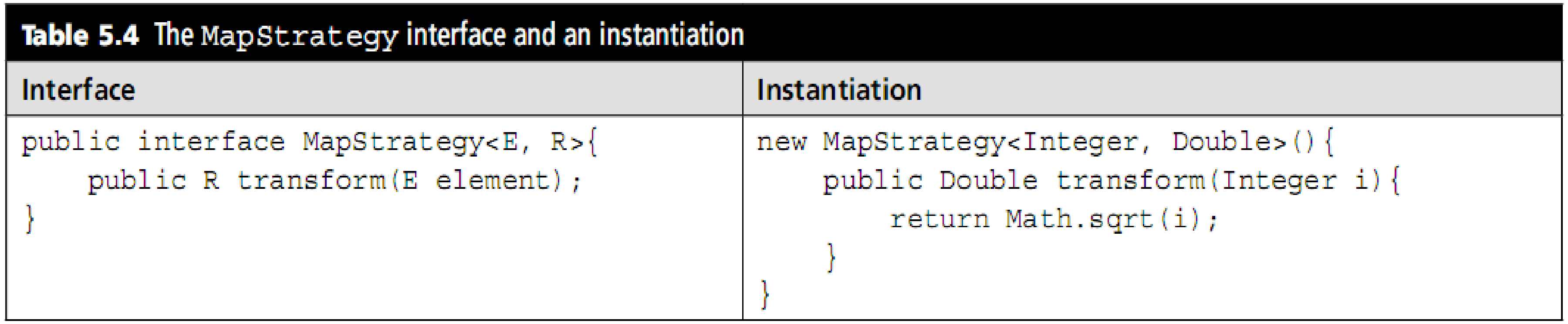


**Table 5.4** The `MapStrategy` interface and an instantiation

**Interface**

```
public interface MapStrategy<E, R>{
    public R transform(E element);
}
```

**Instantiation**

```
new MapStrategy<Integer, Double>(){
    public Double transform(Integer i){
        return Math.sqrt(i);
    }
}
```

## Defining Map, Filter, and Reduce in Java (cont'd.)

- Comparing Java versions of map, filter, and reduce to other languages:
  - Functional versions are simple: they accept other functions as arguments, but they are limited to list collections
  - Smalltalk versions are polymorphic over any collections and no more complicated than that of a lambda form
  - Java syntax is slightly more complicated
- Real benefit of Java is the static type checking, which makes the methods virtually foolproof

## C++

- C++ was originally developed by Bjarne Stroustrup at AT&T Bell Labs
- It is a compromise language that contains most of the C language as a subset, plus other features, some object-oriented, some not
- Now includes multiple inheritance, templates, operator overloading, and exceptions

## Basic Elements of C++: Classes, Data Members, and Member Functions

- C++ contains class and object declarations similar to Java
- **Data members**: instance variables
- **Member functions**: methods
- **Derived classes**: subclasses
- **Base classes**: superclasses
- Objects in C++ are not automatically pointers or references
  - Class data type in C++ is identical to the `struct` or record data type of C

## Basic Elements of C++ (cont'd.)

- Three levels of protection for class members:
  - **Public** members are accessible to client code and derived classes
  - **Protected** members are inaccessible to client code but are accessible to derived classes
  - **Private** members are inaccessible to client and to derived classes
- Keywords `private`, `public`, and `protected` establish blocks in class declarations, rather than apply only to individual member declarations (like in Java)

## Basic Elements of C++ (cont’d.)

```
class A{
// A C++ class
    public:
    // all public members here
    protected:
    // all protected members here
    private:
    // all private members here
};
```

## Basic Elements of C++ (cont'd.)

- **Constructors**: initialize objects as in Java
  - Can be called automatically as part of a declaration, as well as in a `new` expression
- **Destructors**: called when an object is deallocated
  - Name is preceded the tilde symbol (~)
  - Required because there is no built-in garbage collection in C++
- Member functions can be implemented outside the declaration by using the **scope resolution operator** `::` after a class name

## Basic Elements of C++ (cont’d.)

- Member functions with implementations in a class are assumed to be **inline**
  - Compiler may replace the function call with the actual code for the function
- Instance variables are initialized after a colon in a comma-separated list between the constructor declaration and body, with initial values in parentheses

```
Complex(double r = 0, double i = 0)
: re(r), im(i) { }
```

## Basic Elements of C++ (cont'd.)

- Constructor is defined with default values for its parameters

```
Complex(double r = 0, double i = 0)
```

  – This allows objects to be created with 0 to all parameters declared and avoids the need to create overloaded constructors

- Example:

```
Complex i(0,1); // i constructed as (0,1)
Complex y; // y constructed as default (0,0)
Complex x(1); // x constructed as (1,0)
```

# Using a Template Class to Implement a Generic Collection

- **Template classes**: used to define generic collections in C++
- Standard template library (STL) of C++ includes several built-in collection classes
- Example: using a C++ `LinkedStack` class, created with the same basic interface as the Java version presented earlier in the chapter
  - The new `LinkedStack` object is automatically created and assigned to the `stack` variable upon the use of that variable in the declaration

## Using a Template Class to Implement a Generic Collection (cont'd.)

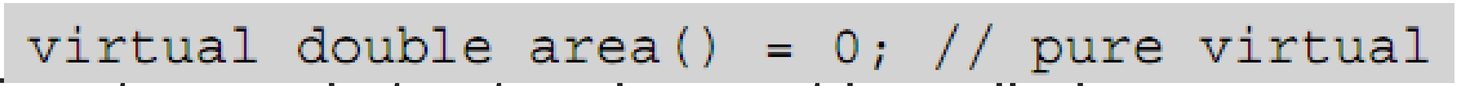
```
#include <iostream>
using std::cout;
using std::endl;
#include "LinkedStack.c"

int main(){
    int number;
    LinkedStack<int> stack;
    for (number = 1; number <= 5; number++)
        stack.push(number);
    cout << "The size is " << stack.size() << endl;
    while (stack.size() != 0)
        cout << stack.pop() << " ";
    cout << endl;
}
```

## Using a Template Class to Implement a Generic Collection (cont'd.)

- Template class is created with keyword `template` in the class header
  - Example: `template <class E>`
- Because this class will utilize dynamic storage for its nodes, it should include a destructor

## Static Binding, Dynamic Binding, and Virtual Functions

- Dynamic binding of member functions is an option in C++, but not the default
  - Only functions defined with the keyword `virtual` are candidates for dynamic binding
- **Pure virtual declaration**: a function declared with a 0 and the keyword `virtual`
  - Example:



```
virtual double area() = 0; // pure virtual
```

  - Function is abstract and cannot be called
  - Renders the containing class abstract
  - Must be overridden in a derived class

## Static Binding, Dynamic Binding, and Virtual Functions (cont'd.)

- Once a function is declared as virtual, it remains so in all derived classes in C++
- Declaring a method as virtual is not sufficient to enable dynamic binding
  - Object must be either dynamically allocated or otherwise accessed through a reference
- C++ offers multiple inheritance using a comma-separated list of base classes
  - Example: `class C : public A, private B {...};`

# Static Binding, Dynamic Binding, and Virtual Functions (cont'd.)

- Multiple inheritance ordinarily creates separate copies of each class on an inheritance path
  - Example: object of class D has two copies of class A

```
class A {...};
class B : public A {...};
class C : public A {...};
class D : public B, public C {...};
```

$$\begin{tikzcd} A & & A \\ B \arrow[u] & & C \arrow[u] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

**Figure 5.10:** Inheritance graph showing multiple inheritance

- This is called **repeated inheritance**

## Static Binding, Dynamic Binding, and Virtual Functions (cont'd.)

- To get a single copy of A in class D, must use the `virtual` keyword, causing **shared inheritance**

```
class A {...};
class B : virtual public A {...};
class C : virtual public A {...};
class D : public B, public C {...};
```

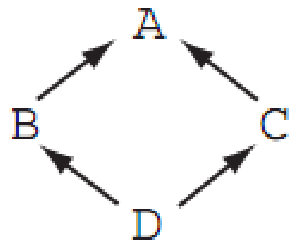

$$\begin{tikzcd} & A & \\ B \arrow[ur] & & C \arrow[ul] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

**Figure 5.11:** Inheritance graph showing shared inheritance

## Design Issues in Object-Oriented Languages

- Object-oriented features represent dynamic rather than static capabilities
  - Must introduce features in a way to reduce the runtime penalty of the extra flexibility
- Inline functions are an efficiency in C++
- Other issues for object-oriented languages are the proper organization of the runtime environment and the ability of a translator to discover optimizations
- Design of the program itself is important to gain maximum advantage of an object-oriented language

## Classes vs. Types

- Classes must be incorporated in some way into the type system
- Three possibilities:
  - Specifically exclude classes from type checking: objects would be typeless entities
  - Make classes type constructors: classes become part of the language type system (adopted by C++)
  - Let classes become the type system: all other structured types are then excluded from the system

## Classes vs. Modules

- Classes provide a versatile mechanism for organizing code
  - Except in Java, classes do not allow the clean separation of implementation from interface and do not protect the implementation from exposure to client code
- Classes are only marginally suited for controlling the importing and exporting of names in a fine-grained way
  - C++ uses a namespace mechanism
  - Java uses a package mechanism

## Inheritance vs. Polymorphism

- Four basic kinds of polymorphism:
  - Parametric polymorphism: type parameters remain unspecified in declarations
  - Overloading (ad hoc polymorphism): different function or method declarations share the same name but have different types of parameters in each
  - Subtype polymorphism: all operations of one type can be applied to another type
  - Inheritance: a kind of subtype polymorphism

## Inheritance vs. Polymorphism (cont'd.)

- **Double-dispatch** (or **multi-dispatch**) problem: inheritance and overloading do not account for binary (or n-ary) methods that may need overloading based on class membership in two or more parameters
- In C++, may need to define a free (overloaded) operator function
- Attempts to solve this problem use **multimethods**: methods that can belong to more than one class or whose overloaded dispatch can be based on class membership of several parameters

# Implementation of Objects and Methods

- Objects are typically implemented exactly as record structures would be in C or Ada
  - Instance variables represent data fields in the structure
  - Example:

```
class Point{
    ...
    public void moveTo(double dx, double dy){
        x += dx;
        y += dy;
    }
    ...
    private double x,y;
}
```

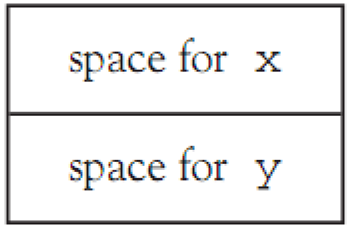
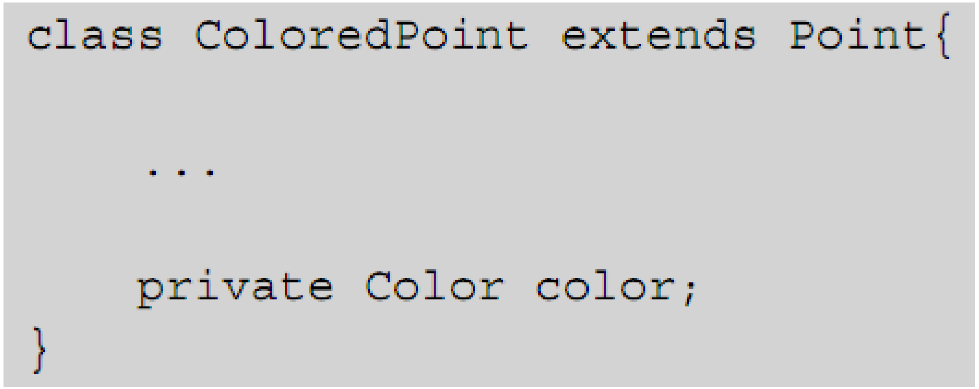


**Figure 5.12:** Space allocated for a C struct

## Implementation of Objects and Methods

- An object of a subclass can be allocated as an extension of the preceding data object, with new instance variables allocated space at the end of the record
  - Example:



```
class ColoredPoint extends Point{

    ...

    private Color color;
}
```

## Implementation of Objects and Methods

- By allocating at the end, the instance variables of the base class can be found at the same offset from the beginning of the allocated space as for any object of the base class

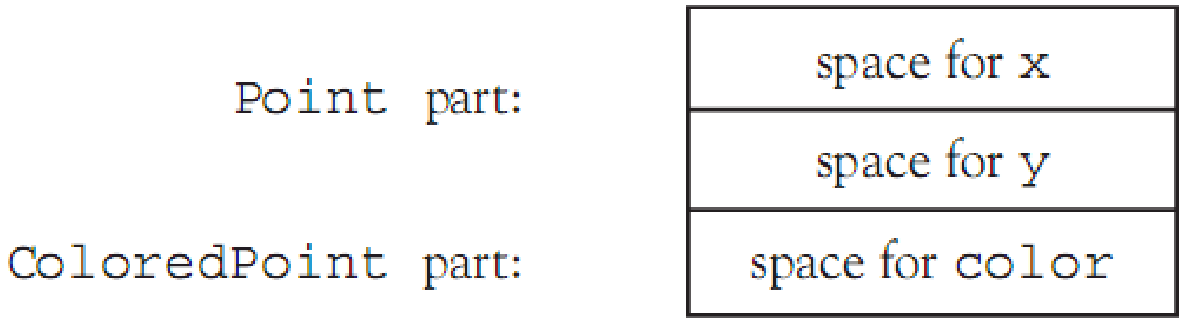


**Figure 5.13:** Allocation of space for an object of class `ColoredPoint`

## Inheritance and Dynamic Binding

- Only space for instance variables is allocated with each object
  - Not provided for methods
- This becomes a problem when dynamic binding is used for methods
  - The precise method to use for a call is not known except during execution
- Possible solution is to keep all dynamically bound methods as extra fields directly in the structures allocated for each object

## Allocation and Initialization

- Object-oriented languages such as C++ maintain a runtime environment in the traditional stack/heap fashion of C
- This makes it possible to allocate objects on either the stack or the heap
  - Java and Smalltalk allocate them on the heap
  - C++ permits an object to be allocated either directly on the stack or as a pointer
- Smalltalk and Java have no explicit deallocation routines but use a garbage collector
  - C++ uses destructors