

Chapter 0: Introduction

Data Structures in Java: From Abstract Data Types to the Java Collections Framework

by Simon Gray



Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Introduction

- This chapter offers a quick introduction to many of the concepts that are presented in greater depth throughout the rest of the text
- What you should look for:
 - Terminology: learn to use vocabulary of object-oriented programming and the “container” types
 - Relationships: begin to get a feel for how the terms and concepts are related to one another

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-2

Abstraction

- A tool to manage complexity
- Hide irrelevant details; focus on the features needed to use a thing
- Examples
 - File deletion using icons
 - The brakes on a car
 - Television remote
 - What else?

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-3

Data Type

- **Specification View** of a data type
 - Data values
 - Operations defined on those values
 - ⇒ Abstract Data Type (ADT) – *note use of abstraction!*
- **Implementation View** of a data type
 - Language-specific representation for the values
 - Implementation of the operations
 - ⇒ Implementation of an ADT

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-4

Algorithm

- A finite sequence of steps that solves a well-defined problem
- Algorithm development (have a plan!):
Specification ⇒ Pseudocode ⇒ Implementation ⇒ Test
- ADT “operations” are algorithms
- Algorithm “cost” – a basis for evaluating and comparing algorithms
 - Time complexity
 - Space complexity

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-5

Object-Oriented Programming

- Model the structure and behavior of a system through the interaction of software objects that represent entities from the problem domain
 - Intuitively, the model “looks” and “behaves” like the system in the problem domain
- Class ⇒ Data Type
 - Data fields ⇒ Values
 - Methods ⇒ Operations
- Information hiding
 - Reveal only those methods you want a client to see ⇒ API
 - Hide implementation details
 - ⇒ abstraction (again!)

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-6

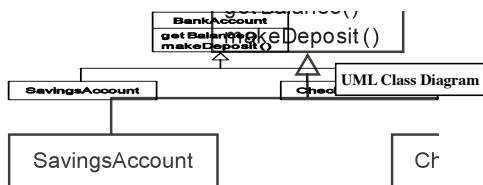
Code Reuse: Classes

- **Inheritance** defines an “is a kind of” relationship between child/subclass and parent/superclass
 - A `TemperatureMonitor` “is a kind of” `Monitor`
 - A subclass inherits capabilities from its superclass
- **Composition** defines a “has a” relationship
 - A `BankAccount` “has a” `String` field to store the client’s name; an `Address` field to store the client’s address, etc.
 - One class has (is composed of) one or more other classes

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0.7

Inheritance Example

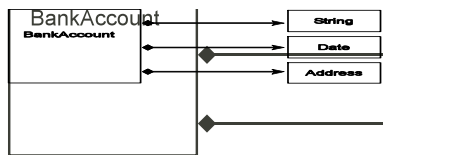


`SavingsAccount` and `CheckingAccount` are kinds of `BankAccount`. Where is the code reuse? `SavingsAccount` and `CheckingAccount` inherit the `getBalance()` and `makeDeposit()` methods from `BankAccount`

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0.8

Composition Example



A `BankAccount` can have a

- `String` field to store the client’s name
- `Date` field to store the account’s creation date
- `Address` field to store the client’s address

UML Class Diagram

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0.9

Unified Modeling Language (UML)

- A *visual* notation to describe the components of a system and their relationships
- **Class diagram**: a static picture of some of the classes in a system and their relationships to one another
- **Object diagram**: a snapshot of some objects in the system showing their values and relationships to other objects
- **Sequence diagram**: shows the sequence of method calls needed to solve a problem (an algorithm)

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-10

Code Reuse: Methods

- A type-specific method is tied to a single type
 - If you want to support more than one type, you need another instance of the method specific for that type
- A generic method is not tied to a single type
 - To apply the method to a different type provide that type through a type argument (type parameter)

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-11

Generic Methods: An Example

Listing 0.1 A swap() method to swap Rational objects in an array.

```
1 void swap( Rational[] theArray, int i, int j ) {  
2     Rational t = theArray[i];  
3     theArray[i] = theArray[j];  
4     theArray[j] = t;  
5 }
```

Note the red entries!

Listing 0.2 A generic swap() method for an array.

```
1 <T> void swap( T[] theArray, int i, int j ) {  
2     T t = theArray[i];  
3     theArray[i] = theArray[j];  
4     theArray[j] = t;  
5 }
```

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-12

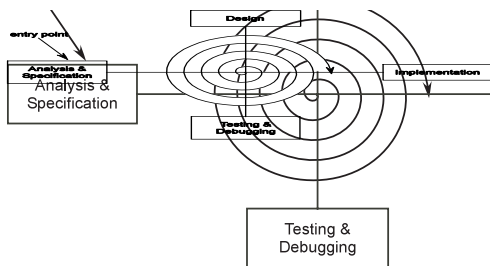
Software Life Cycle

- Analysis and Specification
- Design
- Implementation
- Testing and Debugging
- Maintenance

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-13

Spiral/Incremental Model



Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-14

Software Testing

- So...how do you know your solution is "correct"?
- Proofs of correctness are difficult, so we rely on testing
- When to test?
 - *After the analysis and design are complete.* The design should be critically examined within the context of the problem to be solved. Does it look like the proposed solution will solve the problem?
 - *During the implementation.* As code is completed, its correctness should be tested. This is called **verification** and helps catch problems early and gives you confidence that you are building on a solid code foundation.
 - *After the implementation is complete.* The components should work together as expected and the completed system should meet the client's needs.

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-15

Idea Reuse: Software Design Patterns

- An *abstraction* of a software solution to a commonly encountered software engineering problem
- Avoid reinventing the wheel – adapt a known solution (pattern) to a well-known problem
- Software pattern components: name, context, outline, consequences

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-16

Collection Categories

- Linear Collections
 - List
 - Stack
 - Queue

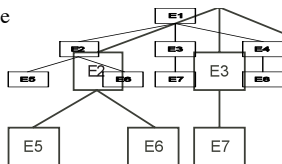


Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-17

Collection Categories

- Hierarchical Collections
 - Tree
 - Binary Tree
 - Binary Search tree

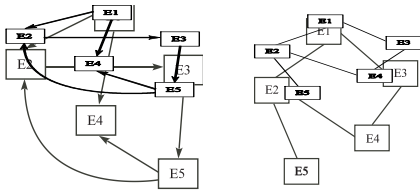


Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-18

Collection Categories

• Graph Collections



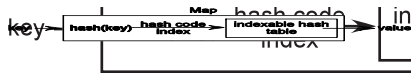
Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-19

Collection Categories

• Nonpositional Collections

- Collection
- Map



Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-20

Collection Operations

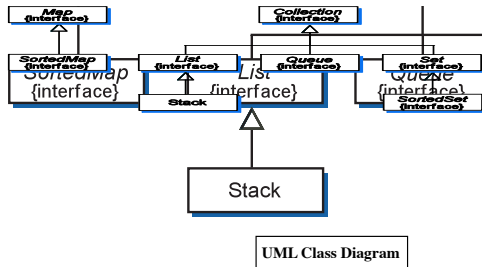
Most collections support the same operations, but may give them different names.

- Add an element
- Remove an element
- Replace an element
- Retrieve an element
- Determine if a collection contains an element
- Get the collection's size
- Determine if a collection is empty
- Traverse a collection
- Determine if two collections are equal
- Clone a collection
- Serialize a collection

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-21

The Java Collections Framework (JCF)



Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

0-22
