Kap.3: Elementsamlinger (Collections)

Mål med kapittelet

- Lære konsepter og terminologi relatert til elementsamlinger
- Kjenne til den grunnleggende struktur for Java Collections API
- Kjenne til det abstrakte designet til elementsamlinger
- Lære å bruke en elementsamling for å løse et problem
- Se grundigere på en tabellimplementering av en elementsamling

Elementsamlinger

- Objekt for lagring og organisering av andre, mindre objekt
- Har bestemte lovlige operasjoner som kan brukes av klientene
- Er dynamiske, dvs. antall element i samlingen kan variere
- To hovedtyper: Lineære (gitt rekkefølgje) og ikke-lineære

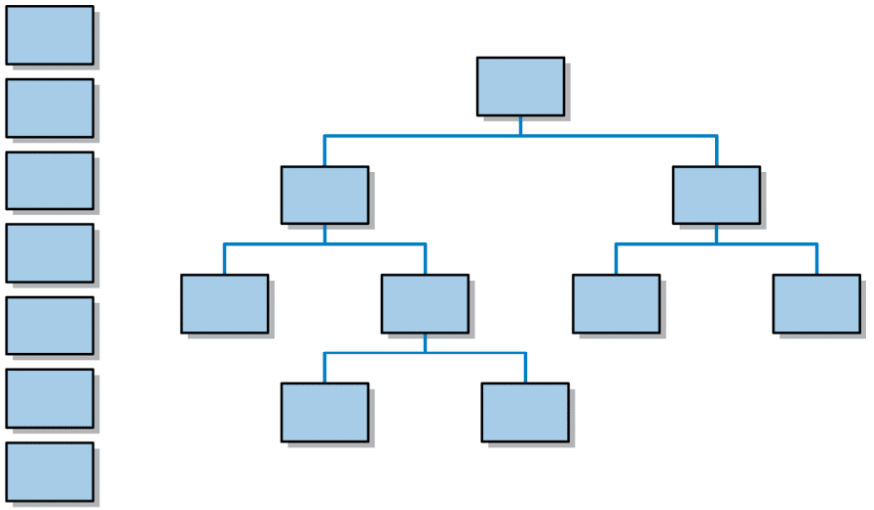

FIGURE 3.1 A linear and a nonlinear collection

Elementsamlinger, forts.

ADT/abstrakt datatype/abstraksjon

- ser bare på det mest vesentlige, det vi har bruk for (data og public-operasjoner)
- ser bort fra andre detaljer

Elementsamlinger ("collections") er ADT-er

- vi definerer grensesnitt for ADT-en (public-metoder) som klienten kjenner.
- den interne implementeringen (hvordan metoder faktisk er implementert og hvilke datastrukturer som benyttes) er ikke kjent for klienten

Elementsamlinger, forts.

Generelt ved utvikling av store programsystem:

- Deler opp i flere mindre delsystem
- Spesifiserer grensesnitt for hvert delsystem
- Objekt/klasser i java kan lett brukes for å lage et delsysystem
- Lett å skille grensesnitt fra implementeringen (koding av data og metoder)

Elementsamlinger, forts.

- Elementer innen en elementsamling er vanligvis organisert basert på:
  - den orden de legges til i samlingen på
  - en eller annen implisitt relasjon mellom elementene
- Eksempelvis – en liste av folk holdes i alfabetisk rekkefølge på navn, eller i den rekkefølge de legges inn i listen
- Spørsmålet om hvilken type elementsamling en skal bruke vil som oftest være avhengig av hvilke oppgaver en ønsker utført på elementsamlingen

Abstraksjon

- En abstraksjon gjemmer enkelte detaljer i enkelte tilfeller
- Tilbyr en måte for å håndtere kompleksiteten i større datasystem
- En elementsamling, på lik linje med alle andre vel definerte objekt, er en abstraksjon
- Vi ønsker å separere interfacen til en elementsamling (hvordan vi interagerer med) fra de underliggende detaljene for hvordan vi velger å implementere den

Elementsamling og Interface

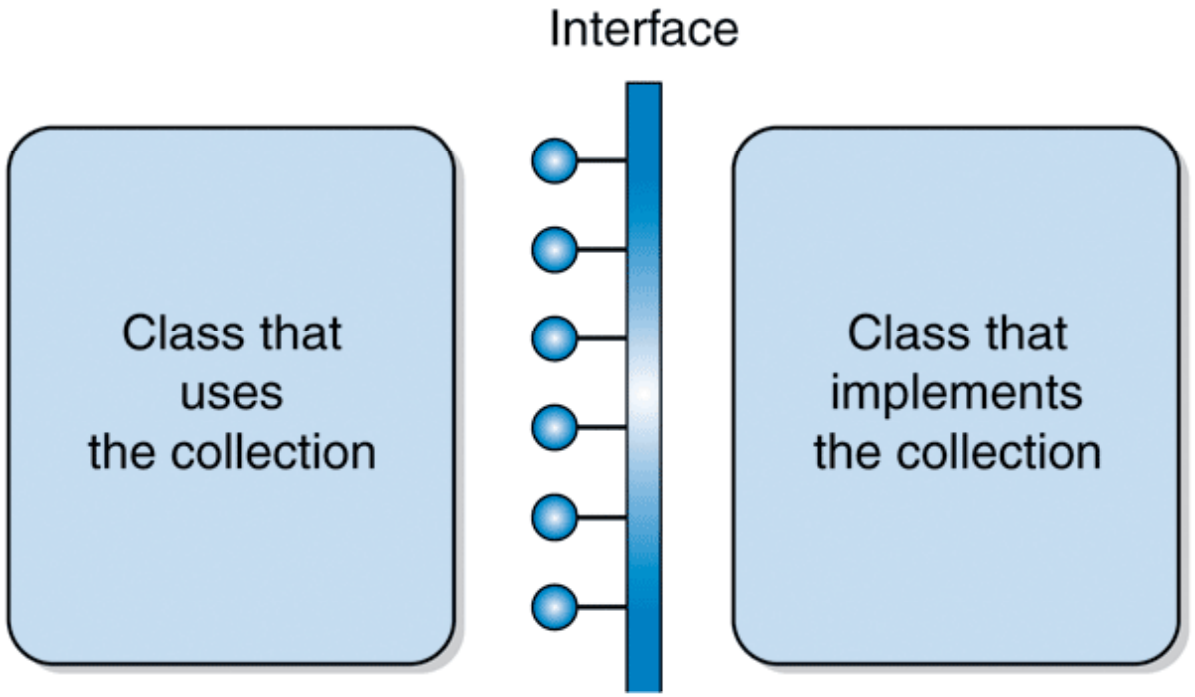

FIGURE 3.2 A well-defined interface masks the implementation of the collection

HØGSKOLEN I BERGEN

Ting rundt elementsamlinger

- For hver elementsamling vi skal jobbe med skal vi se på:
  - Hvordan opererer elementsamlingen konseptuelt?
  - Hvordan definerer vi dens *interface* formelt?
  - Hvilke typer av problem kan den hjelpe oss til å løse?
  - På hvilke måter kan vi implementere den?
  - Hva er fordeler og kostnader ved hver implementering?

HØGSKOLEN I BERGEN

Termer, definisjoner

- Datatype:
  - En gruppe av verdier og de operasjoner som er definert for disse verdiene
- Abstrakt datatype:
  - En datatype hvis verdier og operasjoner ikke er fullstendig definert i et programmeringsspråk
- Datastruktur:
  - Programmeringsstruktur som brukes for å implementere en samling

HØGSKOLEN I BERGEN

Java Collections API

- **API**
  - Application Programming Interface
- … er et sett av klasser som representerer noen spesifikke typer av elementsamlinger på forskjellige måter
- … er en del av det større klassebiblioteket som kan brukes av ethvert Java program
- Skal se på de passende klasser i Java Collections API etter hvert som vi ser på forskjellige former for elementsamlinger

HØGSKOLEN I BERGEN

Elementsamling Stabel

- En stabel organiserer elementer etter et prinsipp om Sist Inn, Først Ut (Last In, First Out) – LIFO
- Kan sammenlignes med en stabel av bøker, tallerkener osv…
- Nye element kan kun plasseres på toppen av stabelen
- Element kan kun fjernes fra toppen av stabelen

Altså: Kan kun utføre operasjoner på topp av stabel

HØGSKOLEN I BERGEN

Konseptuel skisse for stabel

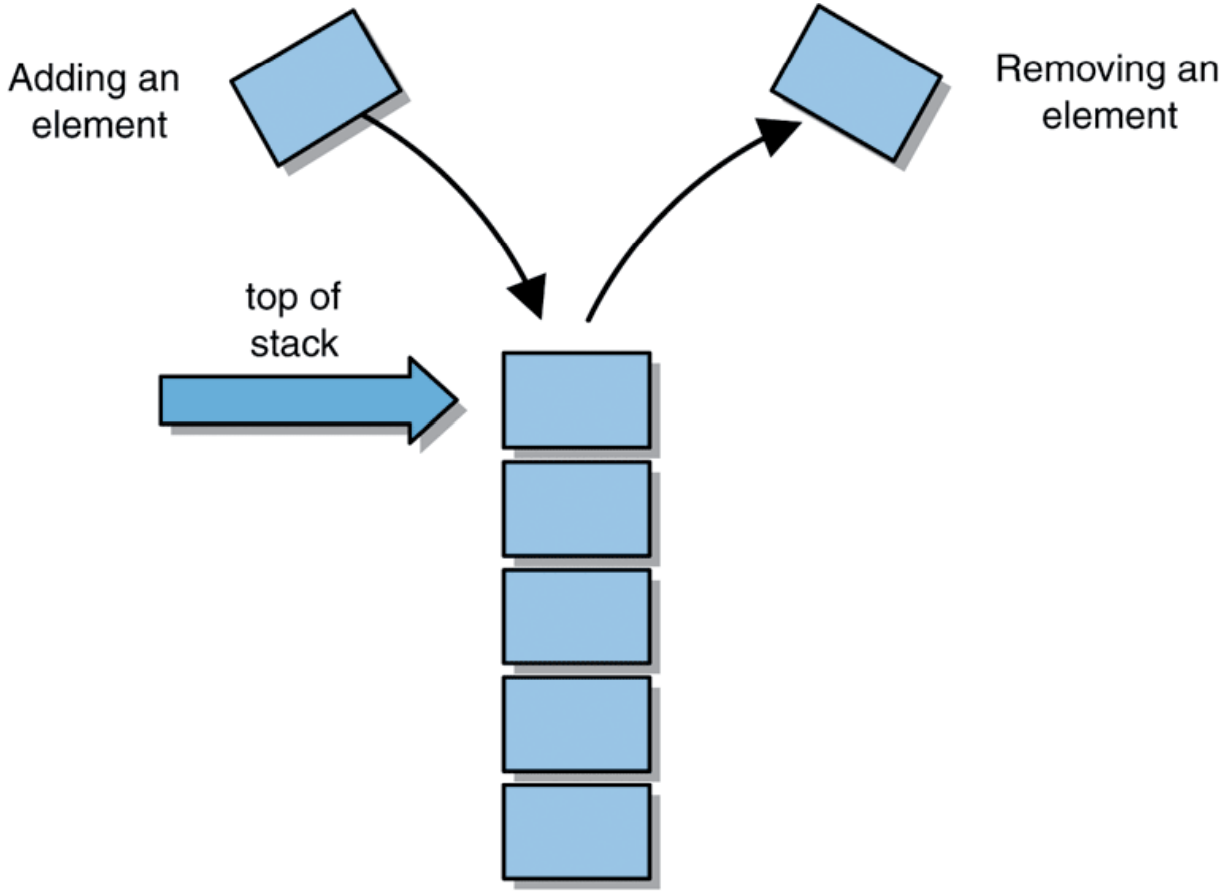

FIGURE 3.3 A conceptual view of a stack

HØGSKOLEN I BERGEN

Operasjoner på samlinger

- Hver samling har et sett av operasjoner som definerer hvordan vi interagerer med dem
- Vanlige operasjoner som dekkes:
  - legge til et eller flere element
  - fjerne et eller flere element
  - bestemme samlingens størrelse

Operasjoner på en stabel

Operation	Description
`push`	Adds an element to the top of the stack.
`pop`	Removes an element from the top of the stack.
`peek`	Examines the element at the top of the stack.
`isEmpty`	Determines if the stack is empty.
`size`	Determines the number of elements on the stack.

FIGURE 3.4 The operations on a stack

Hva kan vi så samle i stabelen?

HØGSKOLEN I BERGEN

- Kanskje vi kunne implementere stabelen på nytt hver gang det er behov for bruk på en ny datatype?
  - Nei, dette er lite effektivt!
- Bedre: opprett en datasamling på et slikt vis at denne kan gjenbrukes av forskjellige datatyper
- Må ha funksjonalitet for sjekk av datatyper og kompatibilitet

- Vi skal her se på muligheter gjennom bruk av teknikker som Arv og Polymorfisme

HØGSKOLEN I BERGEN

Inheritance

- In object-oriented programming, *inheritance* means to derive a class from the definition of another class
- In general, creating a new class via inheritance is faster, easier, and cheaper
- Inheritance is at the heart of software reuse

HØGSKOLEN I BERGEN

Arv

- Begrepet *klasse* kommer fra ideen om å klassifisere grupper av objekt med like karakteristikker
- For eksempel, alle pattedyr deler enkelte karakteristikker som:
  - Varmblodig
  - Har hår
  - Føder levende barn

Arv, forts…

- Modellert for programmering vil klassen `Mammal` (pattedyr) ha variabler og metoder som beskriver hhv. tilstand og adferd til pattedyr
- Fra denne klassen kan vi så avlede klassen `Horse`
- `Class Horse` vil arve alle variabler og metoder som er inneholdt i klassen `Class Mammal`
- `Class Horse` kan også definere egne variabler og metoder i tillegg til de som arves – og en kan også i denne subklassen foreta nye implementeringer av metoder som er arvet fra superklassen

Arv, forts…

- Klassen som brukes for å avlede en annen klasse fra gjennom arv kalles for *foreldreklasse* eller *superklasse*
- Gjennom slik arv etableres det et is-a forhold mellom to klasser.
  - Eks: `horse` is-a `mammal`
- Den avledete klassen kalles gjerne for *subklasse*. En subklasse kan på sin side være foreldreklasse for andre subklasser (av denne igjen).
- Gjennom arv kan en følgelig utvikle *klassehierarkier*

HØGSKOLEN I BERGEN

Eksempel på et klassehierarki

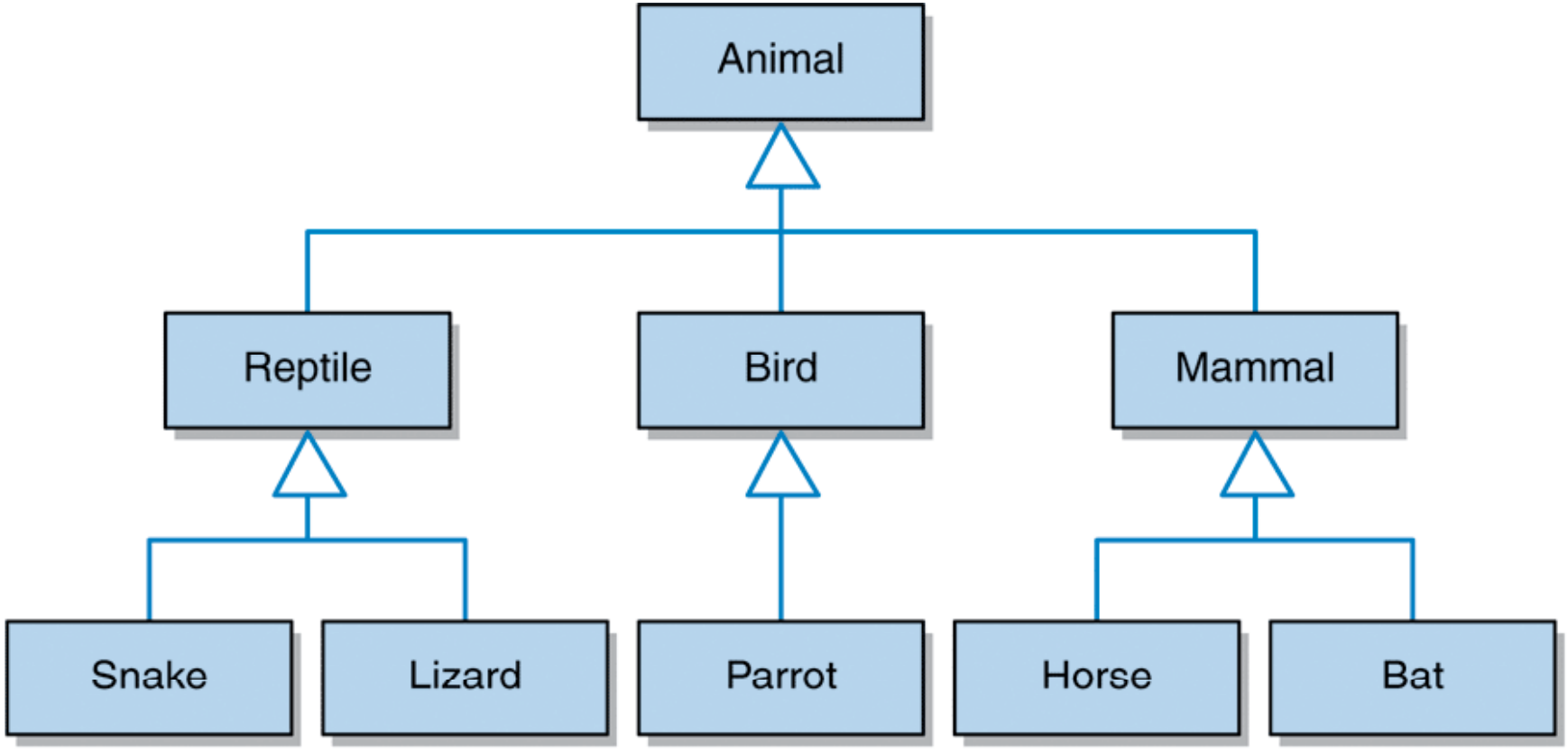

FIGURE 3.5 A UML class diagram showing a class hierarchy

Arv og `object`-klassen

HØGSKOLEN I BERGEN

- I Java er alle klassene avledet ut fra `Object`-klassen
- `Object`-klassen tilbyr en `toString` metode som ofte overskrives av dens sub-klasser
- `Object`-klassen tilbyr også en `equals` metode som tester hvorvidt to objekter er like (

- The `Object` class provides an `equals` method that tests to see if two objects are aliases and is also often overridden

HØGSKOLEN I BERGEN

Polymorphism

- The term *polymorphism* can be defined as "having many forms"
- A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time
- The specific method invoked by a polymorphic reference can change from one invocation to the next

HØGSKOLEN I BERGEN

Polymorphism

- Polymorphism results in the need for *dynamic binding*
- Usually binding of a call to the code occurs at compile time
- Dynamic binding means that this binding cannot occur until run time

Polymorphism

- In Java, a reference that is declared to refer to an object of a particular class can also be used to refer to an object of any class related to it by inheritance
- For example:

```
Mammal pet;
Horse secretariat = new Horse();
pet = secretariat;  // a valid assignment
```

- This is polymorphism as a result of inheritance

HØGSKOLEN I BERGEN

Polymorfisme til det ekstreme, et problem..

- Using polymorphism via inheritance, a reference of type Object can refer to any object of any type
- Thus we could store Object references in our stack allowing us to store objects of any type
- This creates some interesting problems:

```
Animal[ ] creatures = new Mammal[10];
creatures[1] = new Reptile();
```

should not work but it does compile!

HØGSKOLEN I BERGEN

Generiske datatyper

- Bruk av Object klassen gir oss ikke en type-sikker løsning for vår samling
- Java gir oss derimot mulighet til å definere en klasse basert på en *generisk datatype*
- Dette betyr at vi kan definere en klasse slik at den lagrer, opererer på, og håndterer objekt hvis type ikke er spesifisert før klassen er instansiert (altså ikke før det aktuelle objektet av klassen er opprettet)

Generiske datatyper

- Tenk at vi har behov for å definere en klasse ved navn Box som skal kunne lagre og håndtere andre objekt

```
class Box<T>
{
  // declarations and code that manage
  // objects of type T
}
```

Generiske datatyper

- Hvis vi så ønsker å instansiere et objekt av klassen `Box` til å holde objekter av klassen `Ting`

```
Box<Ting> box1 = new Box<Ting>();
```

- Men vi kan også instansiere et objekt av klassen `Box` til å holde objekter av klassen `Tang``Box<Tang>`

```
Box<Tang> box2 = new Box<Tang>();
```

HØGSKOLEN I BERGEN

Generiske datatyper

- En generisk datatype som T kan ikke instansieres
- Den (spesifikke) datatypen som tilordnes ved instansiering tar plassen til den generiske datatypen T alle steder den brukes innen deklareringer i klassen
- Vi skal bruke denne generiske fremgangsmåten på dette kurset når vi har behov for å definere elementsamlinger

HØGSKOLEN I BERGEN

Java Interface

- Programkonstruksjonen i Java kalt *interface* tilbyr en hendig måte for å definere operasjoner på en elementsamling
- En *Java interface* definerer et sett av abstrakte metoder (kun metodehoder – ingen metodekropp) som en klasse må implementere for å kunne implementere det ønskede interface
- Vi kan si at *Java interface* tilbyr en måte for å gi formelle deklarasjoner som en klasse må svare til (et gitt sett av meldinger – metodekall)

The StackADT interface

HØGSKOLEN I BERGEN

- Skal nå lage en slik ADT (Abstract Data Type) for stabelen (”stacken”)
- Kan så i ettertid velge hvordan vi vil implementere denne….

The StackADT interface

HØGSKOLEN I BERGEN

```
/**
 *  @author Lewis and Chase
 *
 *  Defines the interface to a stack data structure.
 */
package jss2;

public interface StackADT<T>
{
  /**  Adds one element to the top of this stack.
   *   @param element element to be pushed onto stack
   */
  public void push (T element);

  /**  Removes and returns the top element from this stack.
   *   @return T element removed from the top of the stack
   */
  public T pop();

  /**  Returns without removing the top element of this stack.
   *   @return T element on top of the stack
   */
  public T peek();
```

The StackADT interface

```
  /**  Returns true if this stack contains no elements.
   *   @return boolean whether or not this stack is empty
   */
  public boolean isEmpty();

  /**  Returns the number of elements in this stack.
   *   @return int number of elements in this stack
   */
  public int size();

  /**  Returns a string representation of this stack.
   *   @return String representation of this stack
   */
  public String toString();
}
```

UML definisjon av SetADT<T> interface

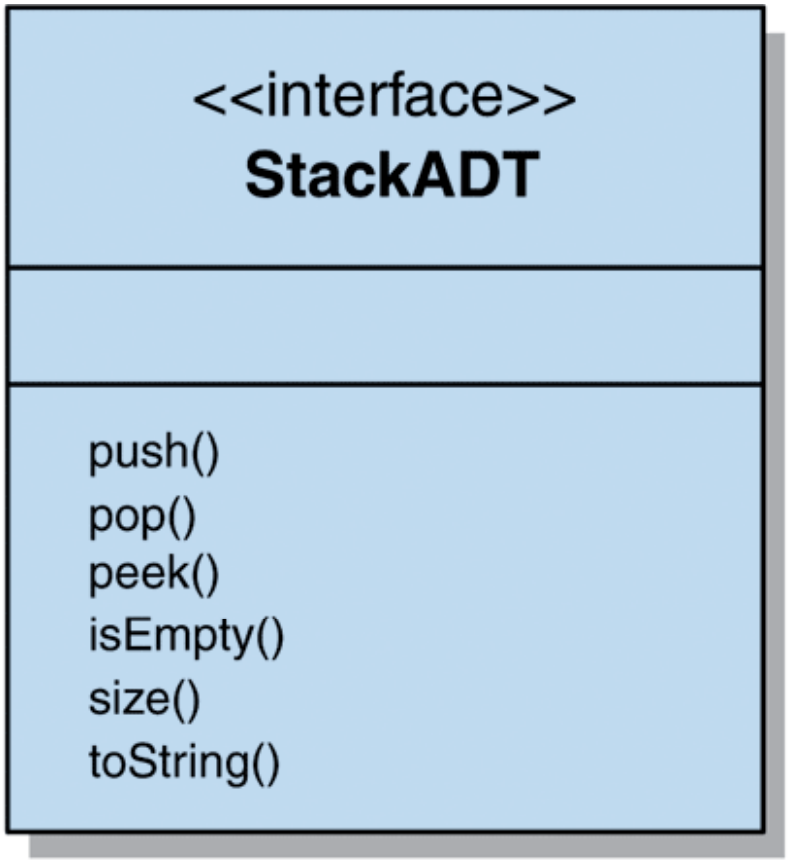

FIGURE 3.6 The `StackADT` interface in UML

HØGSKOLEN I BERGEN

Bruk av stabler

- Stabler er spesielt nyttig når vi skal løse enkelte typer problem
- Tenk på operasjonen undo som vi finner i det meste av applikasjoner
  - Holder orden på de siste utførte operasjonene i reversert rekkefølge (har den siste tilgjengelig først)

Postfix uttrykk

- La oss se på et program som bruker en stabel for å evaluere *postfix* uttrykk
- Innen et *postfix* uttrykk kommer operatoren etter dens to operander (post – operator etter)
- Den normale måten å parse et matematisk uttrykk er *infix* (in – operator mellom), og da gjerne med bruk av paranteser for å styre operatorer og deres presedens:

  (3 + 4) * 2

- Det samme i postfix:

  3 4 + 2 *

Postfix uttrykk

- For å evaluere et postfix uttrykk:
  - Scan/pars fra venstre mot høyre, bestem hvorvidt neste element er en operator eller en operand
  - Hvis operand: push den på stabelen
  - Hvis operator: pop de to øverste elementene av stabelen, utfør operasjonen (gitt av operator) på disse to, og push resultatet tilbake på stabelen
- Til sist vil vi sitte igjen med en verdi i stabelen, og det er svaret (verdien) til uttrykket etter evaluering

Stabel for å evaluere postfix uttrykk

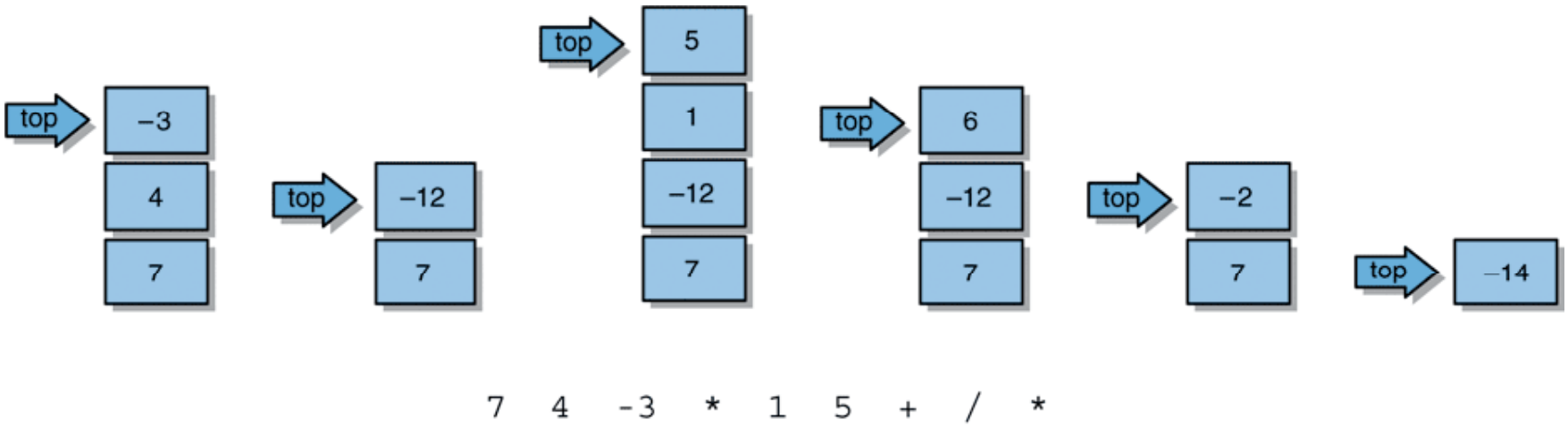

FIGURE 3.7 Using a stack to evaluate a postfix expression

Postfix Expressions

- To simplify the example, let's assume the operands to the expressions are integer literals
- Our solution uses a `LinkedStack`, though any implementation of a stack would suffice

The Postfix class

```
/**
 *  @author Lewis and Chase
 *
 *  Demonstrates the use of a stack to evaluate postfix expressions.
 */
import java.util.Scanner;

public class Postfix
{
  /**
   * Reads and evaluates multiple postfix expressions.
   */
  public static void main (String[] args)
  {
    String expression, again;
    int result;

    try
    {
      Scanner in = new Scanner(System.in);
```

The Postfix class (continued)

```
  do
     {
       PostfixEvaluator evaluator = new PostfixEvaluator();
       System.out.println ("Enter a valid postfix expression: ");
       expression = in.nextLine();

       result = evaluator.evaluate (expression);
       System.out.println();
       System.out.println ("That expression equals " + result);

       System.out.print ("Evaluate another expression [Y/N]? ");
       again = in.nextLine();
       System.out.println();
     }
     while (again.equalsIgnoreCase("y"));
   }
   catch (Exception IOException)
                {
                  System.out.println("Input exception reported");
   }
  }
}
```

The PostfixEvaluator class

```
/**
 *  @author Lewis and Chase
 *
 *  Represents an integer evaluator of postfix expressions. Assumes
 *  the operands are constants.
 */
import jss2.ArrayStack;
import java.util.StringTokenizer;

public class PostfixEvaluator
{
  /** constant for addition symbol */
  private final char ADD = '+';
  /** constant for subtraction symbol */
  private final char SUBTRACT = '-';
  /** constant for multiplication symbol */
  private final char MULTIPLY = '*';
  /** constant for division symbol */
  private final char DIVIDE = '/';
  /** the stack */
  private ArrayStack<Integer> stack;
```

The PostfixEvaluator class (continued)

```
/**
 * Sets up this evalutor by creating a new stack.
 */
public PostfixEvaluator()
{
  stack = new ArrayStack<Integer>();
}

/**
 * Evaluates the specified postfix expression. If an operand is
 * encountered, it is pushed onto the stack. If an operator is
 * encountered, two operands are popped, the operation is
 * evaluated, and the result is pushed onto the stack.
 * @param expr String representation of a postfix expression
 * @return int value of the given expression
 */
public int evaluate (String expr)
{
  int op1, op2, result = 0;
  String token;
  StringTokenizer tokenizer = new StringTokenizer (expr);
```

The PostfixEvaluator class (continued)

```
while (tokenizer.hasMoreTokens())
  {
    token = tokenizer.nextToken();

    if (isOperator(token))
    {
      op2 = (stack.pop()).intValue();
      op1 = (stack.pop()).intValue();
      result = evalSingleOp (token.charAt(0), op1, op2);
      stack.push (new Integer(result));
    }
    else
      stack.push (new Integer(Integer.parseInt(token)));
  }

  return result;
}
```

The PostfixEvaluator class (continued)

```
/**
 * Determines if the specified token is an operator.
 * @param token String representing a single token
 * @return boolean true if token is operator
 */
private boolean isOperator (String token)
{
  return ( token.equals("+") || token.equals("-") ||
           token.equals("*") || token.equals("/") );
}

/**
 * Peforms integer evaluation on a single expression consisting of
 * the specified operator and operands.
 * @param operation operation to be performed
 * @param op1 the first operand
 * @param op2 the second operand
 * @return int value of the expression
 */
private int evalSingleOp (char operation, int op1, int op2)
{
  int result = 0;
```

The PostfixEvaluator class (continued)

```
  switch (operation)
   {
     case ADD:
       result = op1 + op2;
       break;
     case SUBTRACT:
       result = op1 - op2;
       break;
     case MULTIPLY:
       result = op1 * op2;
       break;
     case DIVIDE:
       result = op1 / op2;
   }

   return result;
  }
}
```

A UML class diagram for the postfix expression program

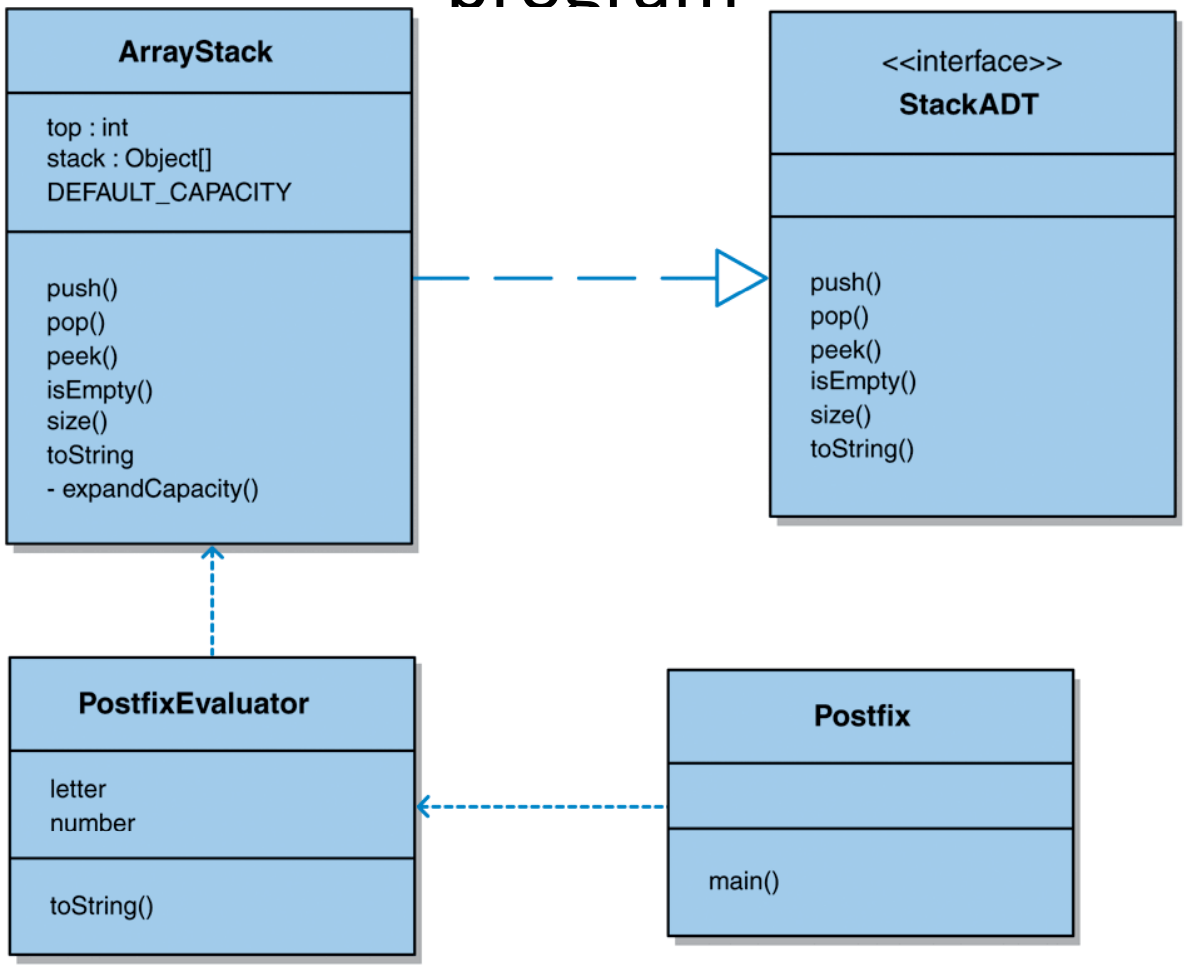

FIGURE 3.8 UML class diagram for the postfix expression evaluation program

Exceptions

- Collections must always manage problem situations carefully
- For example: attempting to remove an element from an empty stack
- The designer of the collection determines how it might be handled
- Our implementation provides an `isEmpty` method, so the user can check beforehand
- And it throws an exception if the situation arises, which the user can catch

Exceptions

- Our stack can be empty, can it ever be full?
- Intuitively, the underlying data structure (array, linked structure, etc.) could run out of memory
- The conceptual stack collection cannot be full
- Thus managing capacity is an issue for our stack implementation

HØGSKOLEN I BERGEN

Managing Capacity

- An array has a particular number of cells when it is created – its capacity
- So the array's capacity is also the stack’s capacity
- What do we do when the stack is full and a new element is added?
  - We could throw an exception
  - We could return some kind of status indicator
  - We could automatically expand the capacity

Managing Capacity

- The first two options require the user of the collection to be on guard and deal with the situation as needed
- The third option is best, especially in light of our desire to separate the implementation from the interface
- The capacity is an implementation problem, and shouldn't be passed along to the user unless there is a good reason to do so

Exceptions

- Problems that arise in a Java program may generate exceptions or errors
- An exception is an object that defines an unusual or erroneous situation
- An error is similar to an exception, except that an error generally represents an unrecoverable situation

HØGSKOLEN I BERGEN

Exceptions

- A program can be designed to process an exception in one of three ways:
  - Not handle the exception at all
  - Handle the exception where it occurs
  - Handle the exception at another point in the program

Exceptions

- If an exception is not handled at all by the program, the program will produce an exception message and terminate
- For example:

Exception in three “main” java.lang.ArithmeticException: / by zero at Zero.main (Zero.java:17)

- This message provides the name of the exception, description of the exception, the class and method, as well as the filename and line number where the exception occurred

Exceptions

- To handle an exception when it is thrown, we use a *try statement*
- A try statement consists of a try block followed by one or more catch clauses

```
try
{
    // statements in the try block
}
catch (IOException exception)
{
    // statements that handle the I/O problem
}
```

Exceptions

- When a try statement is executed, the statements in the try block are executed
- If no exception is thrown, processing continues with the statement following the try statement
- If an exception is thrown, control is immediately passed to the first catch clause whose specified exception corresponds to the class of the exception that was thrown
- If an exception is not caught and handled where it occurs, control is immediately returned to the method that invoked the method that produced the exception

Exceptions

- If that method does not handle the exception (via a try statement with an appropriate catch clause) then control returns to the method that called it
- This process is called propagating the exception
- Exception propagation continues until the exception is caught and handled or until it is propagated out of the main method resulting in the termination of the program

The `ArrayStack` Class

HØGSKOLEN I BERGEN

- Now let's examine an array-based implementation of a stack
- We'll make the following design decisions:
  - maintain an array of generic references
  - the bottom of the stack is at index 0
  - the elements of the stack are in order and contiguous
  - an integer variable `top` stores the index of the next available slot in the array
- This approach allows the stack to grow and shrink at the higher indexes

An array implementation of a stack

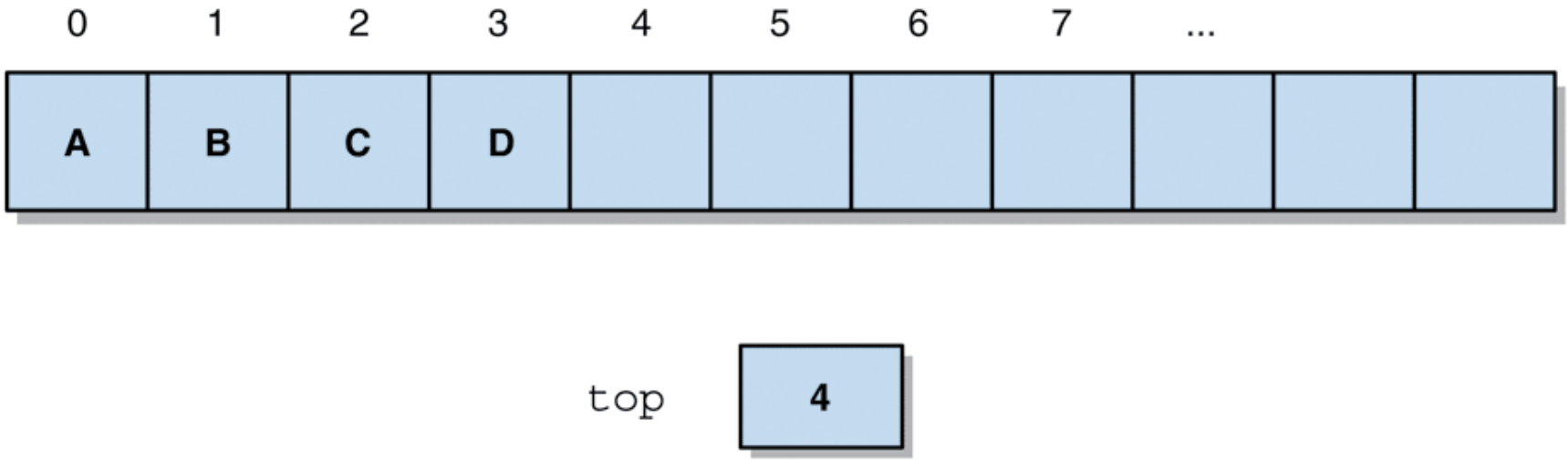

FIGURE 3.10 An array implementation of a stack

ArrayStack

```
/**
*  @author Lewis and Chase
*
*  Represents an array implementation of a stack.
*/
package jss2;
import jss2.exceptions.*;
public class ArrayStack<T> implements StackADT<T>
{
  /**
   * constant to represent the default capacity of the array
   */
  private final int DEFAULT_CAPACITY = 100;
  /**
   * int that represents both the number of elements and the next
   * available position in the array
   */
  private int top;

  /**
   * array of generic elements to represent the stack
   */
  private T[] stack;
```

ArrayStack (continued)

```
/**
 * Creates an empty stack using the default capacity.
 */
public ArrayStack()
{
  top = 0;
  stack = (T[])(new Object[DEFAULT_CAPACITY]);
}

/**
 * Creates an empty stack using the specified capacity.
 * @param initialCapacity represents the specified capacity
 */
public ArrayStack (int initialCapacity)
{
  top = 0;
  stack = (T[])(new Object[initialCapacity]);
}
```

ArrayStack – the push operation

```
/**
  * Adds the specified element to the top of this stack, expanding
  * the capacity of the stack array if necessary.
  * @param element generic element to be pushed onto stack
  */
 public void push (T element)
 {
   if (size() == stack.length)
     expandCapacity();

   stack[top] = element;
   top++;
 }
```

The stack after pushing element E

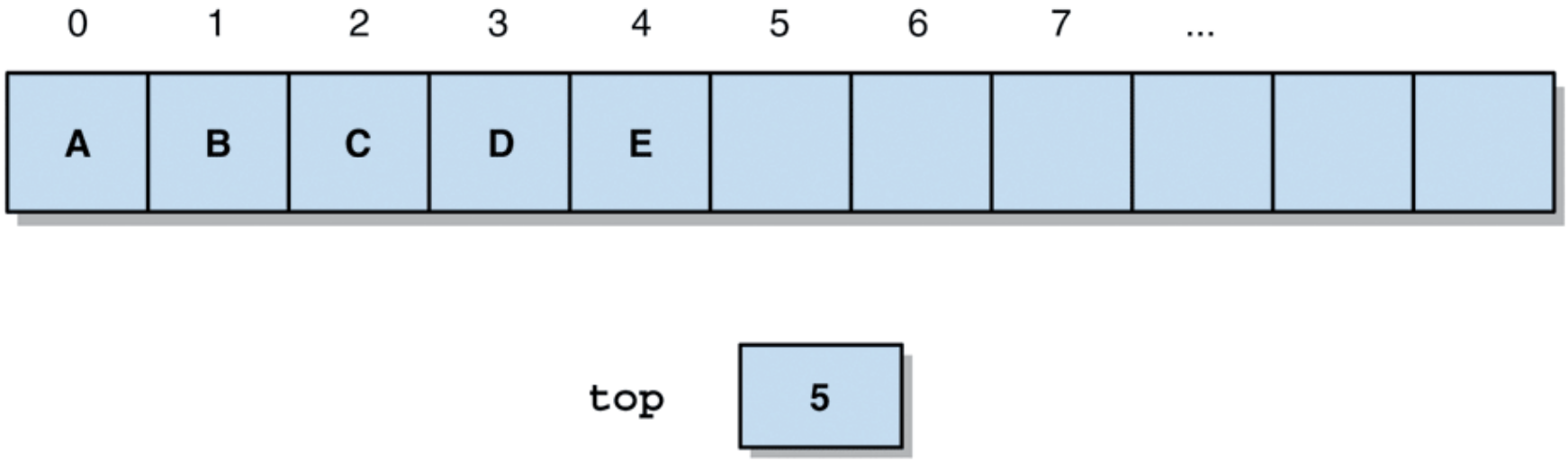

FIGURE 3.11 The stack after pushing element E

ArrayStack – the pop operation

```
/**
 * Removes the element at the top of this stack and returns a
 * reference to it. Throws an EmptyCollectionException if the stack
 * is empty.
 * @return T element removed from top of stack
 * @throws EmptyCollectionException if a pop is attempted on empty stack
 */
public T pop() throws EmptyCollectionException
{
  if (isEmpty())
    throw new EmptyCollectionException("Stack");

  top--;
  T result = stack[top];
  stack[top] = null;

  return result;
}
```

The stack after popping the top element

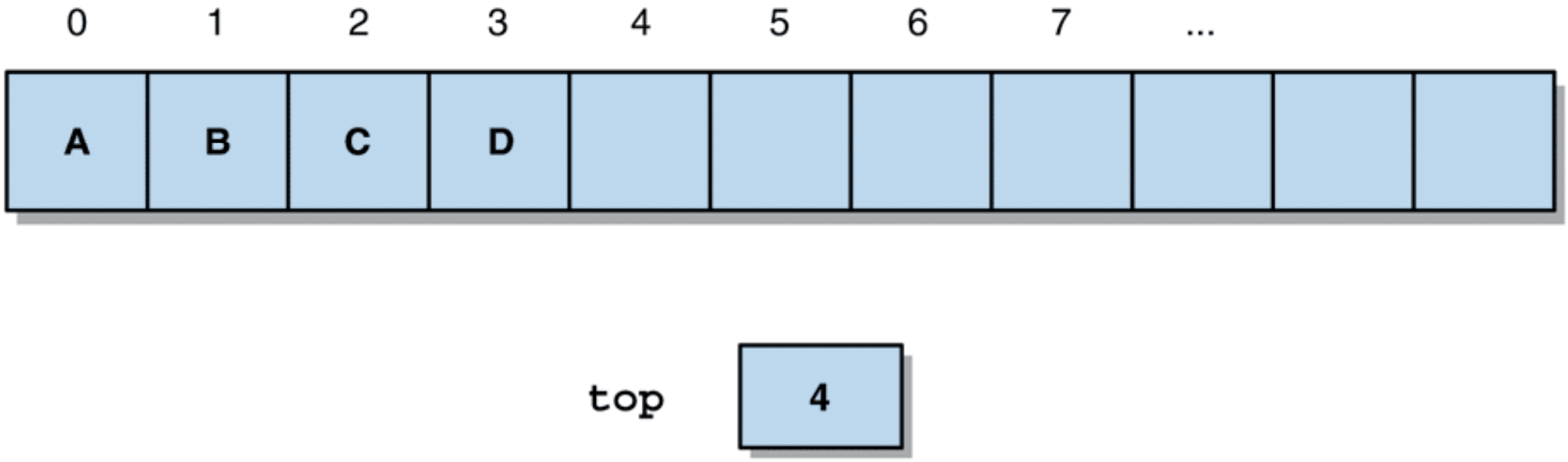

FIGURE 3.12 The stack after popping the top element

ArrayStack – the peek operation

```
/**
  * Returns a reference to the element at the top of this stack.
  * The element is not removed from the stack.  Throws an
  * EmptyCollectionException if the stack is empty.
  * @return T element on top of stack
  * @throws EmptyCollectionException if a peek is attempted on empty stack
  */
 public T peek() throws EmptyCollectionException
 {
   if (isEmpty())
     throw new EmptyCollectionException("Stack");

   return stack[top-1];
 }
```

HØGSKOLEN I BERGEN

Other operations

- The size operation is imply a matter of returning the count
- The isEmpty operation returns true if the count is 0 and false otherwise
- The toString operation concatenates a string made up of the results of the toString operations for each element in the stack

Analysis of Stack Operations

- Because stack operations all work on one end of the collection, they are generally efficient
- The `push` and `pop` operations, for the array implementation are O(1)
- Likewise, the other operations are also O(1)