



DM537

Object-Oriented Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

<http://imada.sdu.dk/~petersk/DM537/>

COLLECTION CLASSES & GENERIC PROGRAMMING

Java Collections Framework

- Java comes with a wide library of *collection classes*
- Examples:
 - `ArrayList`
 - `TreeSet`
 - `HashMap`
- idea is to provide well-implemented standard ADTs
- your own ADTs can build upon this foundation
- collection classes store arbitrary objects
- all collection classes implement `Collection` or `Map`
- thus, simple and standardized interface across different classes

Generic Types (revisited)

- type casts for accessing elements are unsafe!
- solution is to use *generic types*
- instead of using an array of objects, use array of some type E
- Example:

```
public class MyArrayList<E> implements List<E> {  
    ...  
    private E[] data;  
    ...  
    public E get(int i) {  
        return this.data[i];  
    }  
}
```

Generic Programming

- the use of generic types is referred to as *generic programming*
- generic types can and should be used:
 - by the user of collection classes
 - Example: `List<String> list = new ArrayList<String>();`
 - when implementing ADTs
 - Example: `public class MyCollection<E> ...`
 - when implementing constructors and methods
 - Example: `public E getElement(int index) { ... }`
 - when implementing static functions
 - Example: `public <E> void add(ListNode<E> n, E elem);`

Generic Programming

- when a class has parameter type `<E>`, `E` is used like normal type
- instances of the class are defined by substituting concrete type
- Example: `public class Mine<E> ... Mine<String> mine = ...`
- more than one parameter is possible
- Example: `public interface Map<K,V> ...`
- when defining static function, prefix return type by parameter `<E>`
- inside function, `E` is used like normal type
- Example: `public <E> void add(ListNode<E> n, E elem);`

Generic Programming

- we can define that a parameter type extends some interface/class

- Example:

```
public interface BinTree<E extends Comparable> { ... }
```

- then all types E are usable, that implement Comparable

- using “?” we can define wildcard types

- Example:

```
public boolean addAll(Collection<? extends E> c) { ... }
```

- here, elements can be any type that extends E

- the same works with “? super E”

Collection ADT: Specification

- interface `Collection<E>` specifies standard operations
 - `boolean isEmpty();` // true, if there are no elements
 - `int size();` // returns number of elements
 - `boolean contains(Object o);` // is object element?
 - `boolean add(E e);` // add an element; true if modified
 - `boolean remove(Object o);` // remove an element
 - `Iterator<E> iterator();` // iterate over all elements
 - `boolean addAll(Collection<? extends E> c);` // add all ...
 - `clear, containsAll, removeAll, retainAll, toArray, ...`
- operations make sense both for lists, queues, stacks, sets, ...
- next: interface `Iterator<E>`

Iterator ADT: Specification

- iterate over elements of collections (= data)
- operations defined by interface `Iterator<E>`:

```
public interface Iterator<E> {  
    public boolean hasNext();           // is there another element?  
    public E next();                   // get next element  
    public void remove();              // remove current element  
}
```

- can be used to access all elements of the collection
- order is determined by specification or implementation

Iterator ADT: Example I

- Example (iterate over all elements of an `ArrayList`):

```
ArrayList<String> list = new ArrayList<String>();
```

```
list.add("Hej");
```

```
list.add("med");
```

```
list.add("dig");
```

```
Iterator<String> iter = list.iterator();
```

```
while (iter.hasNext()) {  
    String str = iter.next();  
    System.out.println(str);  
}
```

- no need to iterate over indices `0, 1, ..., list.size()-1`

Extended for Loop

- also called “for each loop”
- iterative over each element of an array or a collection
- Example 1 (summing elements of an array):

```
int[] numbers = new int[] {1, 2, 3, 5, 7, 11, 13};
```

```
int sum = 0;
```

```
for (int n : numbers) {
```

```
    sum += n;
```

```
}
```

- Example 2 (multiplying elements of a list):

```
List<Integer> list = new ArrayList(Arrays.asList(numbers));
```

```
int prod = 1;
```

```
for (int i : list) { prod *= i; }
```

List ADT: Usage

- interface `List<E>` extends `Collection<E>`
- additional operation that make no sense for non-lists (e.g. `get`)
- can be sorted by static method in class `Collections`

- Example:

```
int[] numbers = new int[] {1, 2, 3, 5, 7, 11, 13};
```

```
List<Integer> list = new ArrayList(Arrays.asList(numbers));
```

```
Collections.sort(list);
```

- requires that elements implement `Comparable`
- full signature:

```
public static <T extends Comparable<? super T>> void  
    sort(List<T> list);
```

List ADT: Implementations

- **ArrayList** based on dynamic arrays
 - very good first choice in >90% of applications
- **LinkedList** based on doubly-linked lists
 - has prev member variable pointing to previous list node
 - useful when adding and removing a lot in the middle
 - do not use for **Queue** – use **ArrayDeque** instead!
- **Vector** based on dynamic arrays
 - old implementation, not synchronized – use **ArrayList**!
- **Stack** based on Vector
 - do not use for **Stack** – use **ArrayDeque** instead!

Queue ADT: Specification & Implem.

- interface `Queue<E>` extends `Collection<E>`
- data are arbitrary objects of type `E`
- defines additional operations over `Collection<E>`:
 - `public boolean offer(E e);` // alternative name to add
 - `public E peek();` // return head
 - `public E element();` // alternative name to peek
 - `public E poll();` // remove and return head
- extended again by interface `Deque<E>` providing support for adding AND removing at both ends
- Implementations:
 - `ArrayDeque` – with `offer == offerLast` and `poll == pollFirst`
 - `LinkedList` – only useful, when not a pure `Queue`

Stack ADT: Specification & Implem.

- class `Stack<E>` implements `Collection<E>`
- data are arbitrary objects of type `E`
- defines additional operations over `Collection<E>`:
 - `public E push(E e);` // add on top of stack
 - `public E peek();` // return top element
 - `public E pop();` // remove and return top
 - `public int search(Object o);` // return 1-based index
- superseded by interface `Deque<E>` providing support for adding AND removing at both ends
- Alternative Implementations:
 - `ArrayDeque` – with `push == addFirst` and `pop == removeFirst`

Deque ADT: Specification & Implem.

- interface `Deque<E>` extends `Collection<E>`
- data are arbitrary objects of type `E`
- defines additional operations over `Collection<E>`:
 - `addFirst`, `offerFirst`, `addLast`, `offerLast`
 - `removeFirst`, `pollFirst`, `removeLast`, `pollLast`
 - `getFirst`, `peekFirst`, `getLast`, `peekLast`
- `add*`, `remove*`, `get*` throw exceptions
- `offer*`, `poll*`, `peek*` return special value
- Implementations:
 - `ArrayDeque` – fast and preferred
 - `LinkedList` – only use when more than `Deque` needed

Set ADT: Specification

- interface `List<E>` extends `Collection<E>`
- unordered sequences of objects without duplicates
- no additional operations, as `Collection<E>` already specifies
 - `isEmpty`, `size`, `contains`, `add`, `remove`, ...
- no index-based access to elements, as order undefined
- elements MUST implement `equals` and `hashCode` correctly:
 1. for two elements `e1` and `e2` that are equal, both `e1.equals(e2)` and `e2.equals(e1)` must return `true`
 2. for two elements `e1` and `e2` that are equal, we must have `e1.hashCode() == e2.hashCode()`
 3. for two elements `e1` and `e2` that are NOT equal, both `e1.equals(e2)` and `e2.equals(e1)` must return `false`

Set ADT: Example

- Example (intersecting two sets):

```
int[] n1 = new int[] {1, 2, 3, 5, 7, 11, 13};
```

```
Set<Integer> set1 = new HashSet<Integer>(Arrays.asList(n1));
```

```
int[] n2 = new int[] {1, 3, 5, 7, 9};
```

```
Set<Integer> set2 = new HashSet<Integer>(Arrays.asList(n2));
```

```
Set<Integer> set3 = new HashSet<Integer>(set1);
```

```
set3.retainAll(set2);
```

- retainAll modifies set3, thus we have (informally):
 - set1 == {1, 2, 3, 5, 7, 11, 13}
 - set2 == {1, 3, 5, 7, 9}
 - set3 == {1, 3, 5, 7}

Iterator ADT: Example 2

- Example (iterate over all elements of a `HashSet`):

```
Set<String> set = new HashSet<String>();  
set.add("Hej");  
set.add("hej");  
set.add("Hej");  
Iterator<String> iter = set.iterator();  
while (iter.hasNext()) {  
    String str = iter.next();  
    System.out.println(str);  
}
```

- prints the two strings in some undefined order

Interface Comparator

- allows to specify how to compare elements

```
public interface Comparator<E> {  
    public int compare(E o1, E o2);    // compare o1 and o2  
    public boolean equals(Object obj); // equals other Comparator?  
}
```

- compare behaves like `o1.compareTo(o2)` from `Comparable<E>`
 - `< 0` for `o1` less than `o2`
 - `==0` for `o1` equals `o2`
 - `> 0` for `o1` greater than `o2`
- `Comparable` defines *natural* ordering
- `Comparator` can define additional orderings

Set ADT: TreeSet Implementation

- `TreeSet` implements sets as special sort trees (Red-Black Trees)
- elements are compared to according to natural ordering
- Example: `public class Compi implements Comparator<Integer> {
 public int compare(Integer i1, Integer i2) {
 return i2.compareTo(i1); }
 public boolean equals(Object other) { return false; } } ...`
`TreeSet<Integer> set1 = new TreeSet<Integer>();
set1.add(23); set1.add(42); set1.add(-3);
for (int n : set1) { System.out.print(" "+n); } // -3 23 42`
`TreeSet<Integer> set2 = new TreeSet<Integer>(new Compi());
set2.addAll(set1);
for (int n : set2) { System.out.print(" "+n); } // 42 23 -3`

Set ADT: Implementations

- **HashSet** based on hash tables
 - very good choice if order really does not matter
- **LinkedHashSet** based on hash tables + linked list
 - in addition to hash table keeps track of insertion order
 - useful for keeping algorithms deterministic
- **TreeSet** based on special sort trees
 - implements the **SortedSet<E>** interface
 - useful for ordered sequences without duplicates
 - can use **Comparators** for different orderings
 - also useful when e.g. hash code not available

Map ADT: Specification

- maps work like dictionaries in Python
- interface `Map<K,V>` specifies standard operations
 - `boolean isEmpty();` // true, if there are no mappings
 - `int size();` // returns number of mappings
 - `boolean containsKey(Object key);` // is key mapped?
 - `boolean containsValue(Object value);` // is value mapped?
 - `V get(Object key);` // return mapped value or null
 - `V put(K key,V value);` // add mapping from key to value
 - `Set<K> keySet();` // set of all keys
 - `Collection<V> values();` // collection of all values
 - `Set<Map.Entry<K,V>> entrySet();` // (key,value) pairs
 - `clear, putAll, remove, ...`

Map ADT: Example

- Example (using and modifying a phone directory):

```
Map<String,Integer> dir = new HashMap<String,Integer>();
dir.put("petersk", 65502327); dir.put("bwillis", 55555555);
for (String key : dir.keySet()) {
    System.out.println(key+" -> "+dir.get(key));
}
for (Map.Entry<String,Integer> entry : dir.entrySet()) {
    System.out.println(entry.getKey()+" -> "+entry.getValue());
    entry.setValue(12345678);
}
dir.keySet().remove("bwillis");
System.out.println(dir);    // only petersk is mapped
```


Hash Table

- a hash table uses the `hashCode` method to map objects to `ints`
- objects are stored in an array
- the position of the object is determined by its hash code modulo the length of the array
- Example: if `o` has hash code `10` and array has length `7`,
`o` is stored at position $10 \% 7 == 3$
- more in **DM507 Algorithms and Data Structures**
- efficient for get and put
- assuming that `hashCode` is implemented in a useful way
- if two or more objects have the same hash code, the array stores a list of objects in that position

Map ADT: Implementations

- **HashMap** based on hash tables
 - very good choice if order does not matter
- **LinkedHashMap** based on hash tables + linked list
 - in addition to hash table keeps track of insertion order
 - useful for keeping algorithms deterministic
- **TreeMap** based on special sort trees
 - implements the **SortedMap<K,V>** interface
 - useful for ordered mappings
 - can use **Comparators** for different orderings
 - also useful when e.g. hash code not available
- **Hashtable** based on hash tables
 - old implementation – only use for synchronization

IN & OUTPUT USING STREAMS

Streams

- streams are ADTs for representing input and output
- source for input can e.g. be files, keyboard, network resources
- output can go to e.g. files, terminal, network resources
- four categories of streams in `java.io` package:

	Input	Output
byte	InputStream	OutputStream
character	Reader	Writer

- byte streams are for machine-readable data
 - reading one unit is reading one byte (= 8 bits)
- character streams are for human-readable data
 - reading one unit is reading one character (= 16 bits)
 - readers/writers translate 8-bit files etc. into 16-bit unicode

InputStream ADT: Specification

- data = potentially infinite stream of bytes
- operations are given by the following interface:

```
public interface InputStreamADT {  
    public int available();           // how much more can be read?  
    public void close();             // close the stream  
    public int read();               // next byte of the stream  
    public int read(byte[] b);      // read n bytes into b and return n  
    public int read(byte[] b, int off, int len); // max len from b[off]  
    public long skip(long n);       // skip n bytes  
}
```

- all input byte streams are subclasses of `java.io.InputStream`

InputStream ADT: Example

- Example (reading up to 1024 bytes from a file):

```
InputStream input = new FileInputStream(new File("test.txt"));
```

```
byte[] data = new byte[1024];
```

```
int readSoFar = 0;
```

```
do {
```

```
    readSoFar += input.read(data, readSoFar, 1024-readSoFar);
```

```
} while (input.available() > 0 && readSoFar < 1024);
```

```
input.close();
```

```
System.out.println("Got "+readSoFar+" bytes from test.txt!");
```

- if you think that is horrible ...
- ... you now understand, why we used `java.util.Scanner` 😊

OutputStream ADT: Specification

- data = potentially infinite stream of bytes
- operations are given by the following interface:

```
public interface OutputStreamADT {  
    public void close();           // close the stream  
    public void write(int b);     // write b to the stream  
    public void write(byte[] b); // write b.length bytes from b  
    public void write(byte[] b, int off, int len); // len bytes from b[off]  
    public void flush();         // forces buffers to be written  
}
```

- all output byte streams are subclasses of `java.io.OutputStream`

OutputStream ADT: Example

- Example (copying a file):

```
InputStream in = new FileInputStream(new File("test.txt"));
OutputStream out = new FileOutputStream(new File("test.out"));
int total = 0;
byte[] block = new byte[4096];
while (true) {
    int read = in.read(block);
    if (read == -1) { break; }
    out.write(block, 0, read);
    total += read;
}
in.close();    out.close();
System.out.println("Copied "+total+" bytes from test.txt!");
```


Reader ADT: Specification

- data = potentially infinite stream of characters
- operations are given by the following interface:

```
public interface ReaderADT {  
    public boolean ready();    // input available?  
    public void close();      // close the stream  
    public int read();        // next character of the stream  
    public int read(char[] c); // read n characters into c and return n  
    public int read(char[] c, int off, int len); // max len from c[off]  
    public int read(CharBuffer target); // read into CharBuffer  
    public long skip(long n); // skip n characters  
}
```

- all input character streams are subclasses of `java.io.Reader`

Reader ADT: Example

- Example (reading characters from a file):

```
Reader input = new FileReader(new File("test.txt"));
```

```
StringBuffer buffer = new StringBuffer();
```

```
while (true) {
```

```
    int ch = input.read();
```

```
    if (ch == -1) { break; }
```

```
    buffer.append((char)ch);
```

```
}
```

```
input.close();
```

```
System.out.println("Read the following content:");
```

```
System.out.println(buffer.toString());
```

- less horrible ... but we still prefer `java.util.Scanner` 😊

Writer ADT: Specification

- data = potentially infinite stream of characters
- operations are given by the following interface:

```
public interface WriterADT {  
    public void close();           // close the stream  
    public void write(int c);      // write one character to the stream  
    public void write(char[] c); // write c.length characters  
    public void write(char[] c, int off, int len); // len chars from c[off]  
    public void write(String s); // write s.length() characters  
    public void write(String s, int off, int len); // len chars from s at off  
    public void flush();          // forces buffers to be written  
}
```

- all input character streams are subclasses of `java.io.Writer`

Writer ADT: Example

- Example (copying a text file character by character):

```
Reader in = new FileReader(new File("test.txt"));
```

```
Writer out = new FileWriter(new File("test.out"));
```

```
while (true) {
```

```
    int ch = in.read();
```

```
    if (ch == -1) { break; }
```

```
    out.write(ch);
```

```
}
```

```
in.close();
```

```
out.close();
```

```
System.out.println("Done!");
```

Character vs Byte Streams

- Java has classes to convert between character and byte streams
- characters are converted according to specified char set
- default char set is 16-bit unicode

	Input	Output
byte -> char	InputStreamReader	DataOutputStream
char -> byte	DataInputStream	OutputStreamWriter

- InputStreamReader reads characters from byte stream
- ByteArrayOutputStream can be used to write primitive types + String
- OutputStreamWrite write characters to byte stream
- DataInputStream can be used to read primitive types + String

PrintWriter & PrintStream

- classes that extend `Writer` and `OutputStream`
- add comfortable methods for printing and formatting data
- provide methods such as for example
 - `print` – like in `System.out.print`
 - `println` – like in `System.out.println`
 - `printf` – like in `System.out.printf`
- in fact, `System.out` is an instance of `PrintStream`
- Example (writing comfortably to a file):

```
File file = new File("test.out");    String name = "Peter";  
PrintStream out = new PrintStream(new FileOutputStream(file));  
out.printf("Hej %s! How are you?\n", name);  
out.close();
```

NETWORKING & MULTI-THREADING

Accessing Network Resources

- like `File` represents files, `URL` represents network resources
- Example 1 (downloading course web site into file):

```
URL url = new URL("http://imada.sdu.dk/~petersk/DM537/");
InputStream input = url.openStream();
OutputStream output = new FileOutputStream("dm537.html");
byte[] block = new byte[4096];
while (true) {
    int read = input.read(block);
    if (read == -1) { break; }
    output.write(block, 0, read);
}
input.close(); output.close();
```


Accessing Network Resources

- like `File` represents files, `URL` represents network resources
- Example 2 (downloading course web site into file):

```
URL url = new URL("http://imada.sdu.dk/~petersk/DM537/");
```

```
Reader in = new InputStreamReader(url.openStream());
```

```
PrintStream output = new PrintStream(  
    new FileOutputStream("dm537.html"));
```

```
BufferedReader input = new BufferedReader(in);
```

```
while (true) {  
    String line = input.readLine();  
    if (line == null) { break; }  
    output.println(line);  
}    input.close();    output.close();
```

TCP/IP Sockets

- **URL** provides high-level abstraction
- for general TCP/IP connection, *sockets* are needed
- once socket connection is established, normal byte streams
- client-server model where server waits for client to connect
- for sockets, IP address and port number needed
- Example: IP 130.225.157.85, Port 80 (IMADA web server)
- listening sockets implemented by class **ServerSocket**
- Example: `ServerSocket ss = new ServerSocket(2342);`
- connection between client and server instance of **Socket**
- Example: `Socket sSock = ss.accept();`
`Socket sock = new Socket("127.0.0.1", 2342);`

Example: TCP/IP Server

```
public class MyServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(2343);
        while (true) {
            Socket sock = server.accept();
            InputStream in = sock.getInputStream();
            OutputStream out = sock.getOutputStream();
            while (true) {
                int read = in.read();
                if (read == -1) { break; }
                out.write(Character.toUpperCase((char)read));
            }
        }
    }
}
```

Example:TCP/IP Client

```
public class MyClient {
    public static void main(String[] args) throws IOException {
        Socket sock = new Socket("127.0.0.1", 2343);
        InputStream in = sock.getInputStream();
        OutputStream out = sock.getOutputStream();
        String userInput = new Scanner(System.in).nextLine();
        StringBuffer result = new StringBuffer();
        for (char ch : userInput.toCharArray()) {
            out.write(ch);
            result.append((char)in.read());
        }
        System.out.println(result); } }
```

Example: Simple Chat Server

```
public class ChatServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket server = new ServerSocket(2343);  
        while (true) {  
            Socket sock = server.accept();  
            Scanner in = new Scanner(sock.getInputStream());  
            PrintStream out = new PrintStream(sock.getOutputStream());  
            while (true) {  
                System.out.println(in.nextLine());  
                out.println(new Scanner(System.in).nextLine());  
            }  
        }  
    }  
}
```

Example: Simple Chat Client

```
public class ChatClient {  
    public static void main(String[] args) throws IOException {  
        Socket sock = new Socket("127.0.0.1", 2343);  
        Scanner in = new Scanner(sock.getInputStream());  
        PrintStream out = new PrintStream(sock.getOutputStream());  
        while (true) {  
            out.println(new Scanner(System.in).nextLine());  
            System.out.println(in.nextLine());  
        }  
    }  
}
```

Theory and Practice

- our client-server implementations work fine
- BUT:
 - network connections are not reliable
 - there can be many clients
 - answering queries can be time consuming
- multi-threading can solve these problems
- Idea:
 - create a thread for each client connection
 - the server is immediately responsive
 - starving threads can be disposed of after some timeout

Multi-Threading

- threads can be started by creating instances of `Thread`
- Example (two threads counting up to 1 000 000):

```
public class Counter extends Thread {  
    String name;  
    public Counter(String name) { this.name = name; }  
    public void run() {  
        for (int i=1; i<=1000000; i++) {  
            System.out.printf("%s: %i\n", name, i);  
        }  
    }  
}  
...
```


Multi-Threading

- Example (continued):

...

```
public static void main(String[] args) {  
    Counter c1 = new Counter("Fred");  
    Counter c2 = new Counter("George");  
    c1.start();  
    c2.start();  
}  
}
```

- `start()` creates a new thread and runs the `run()` method

Multi-Threaded Server

```
public class MultiServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket server = new ServerSocket(2343);  
        while (true) {  
            Socket sock = server.accept();  
            new MultiServerHandler(sock).start();  
        }  
    }  
}
```

Multi-Threaded Server

```
public class MultiServerHandler extends Thread {  
    private Socket sock;  
    public MultiServerHandler(Socket sock) {  
        this.sock = sock;  
    }  
    public void run() {  
        try {  
            Scanner in = new Scanner(sock.getInputStream());  
            PrintStream out = new PrintStream(sock.getOutputStream());  
            while (true) { out.println(in.nextLine().toUpperCase()); }  
        } catch (IOException e) {}  
    }  
}
```

THE END