INB370 / INN370 Software Development Lecture 4 — Collections, Generics and Exceptions

Faculty of Science and Technology

Semester 1, 2010

Aims of the Week 4 lecture and practical session

- To appreciate the design of abstract data type *collections* as a basis for writing efficient and maintainable programs
  - Collections as pre-defined compound data types
  - Collections as extendable data types
  - The structure of Java's collections framework
- To understand *generics* as a way of parameterising abstract data types
  - Generics as type-valued parameters
  - Generic types in Java
- To learn how to throw, catch and define *exceptions*
  - Exceptions as recoverable errors
  - Exception types in Java

Part A — Collections

What is a collection?

- A *compound data type* that groups multiple elements of the same type together
  - Encapsulates the data structure needed to store the elements and the algorithms needed to manipulate and access them
- All high-level programming languages implement this concept in the form of homogeneous *arrays*
  - However, an object-oriented language offers the ability to define a more powerful collections framework
- Advantages:
  - Saves reinventing the wheel
  - Provides a common, extendable programming environment
- The `java.util` package contains Java's implementations of basic collections

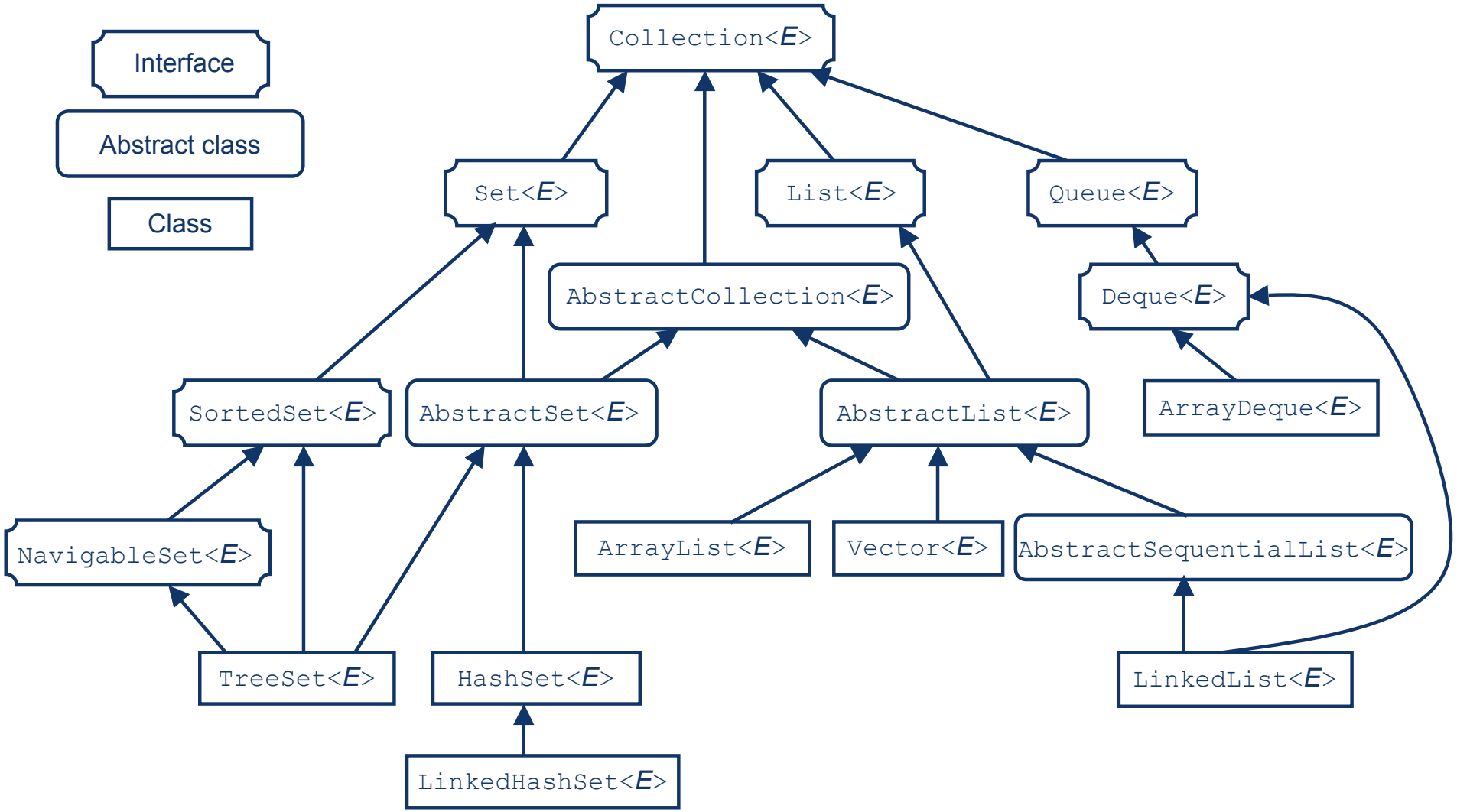
Reference:
D. Watt and D. Brown. *Java Collections: An Introduction to Abstract Data Types, Data Structures and Algorithms*. Wiley, 2001.

A mathematical aside

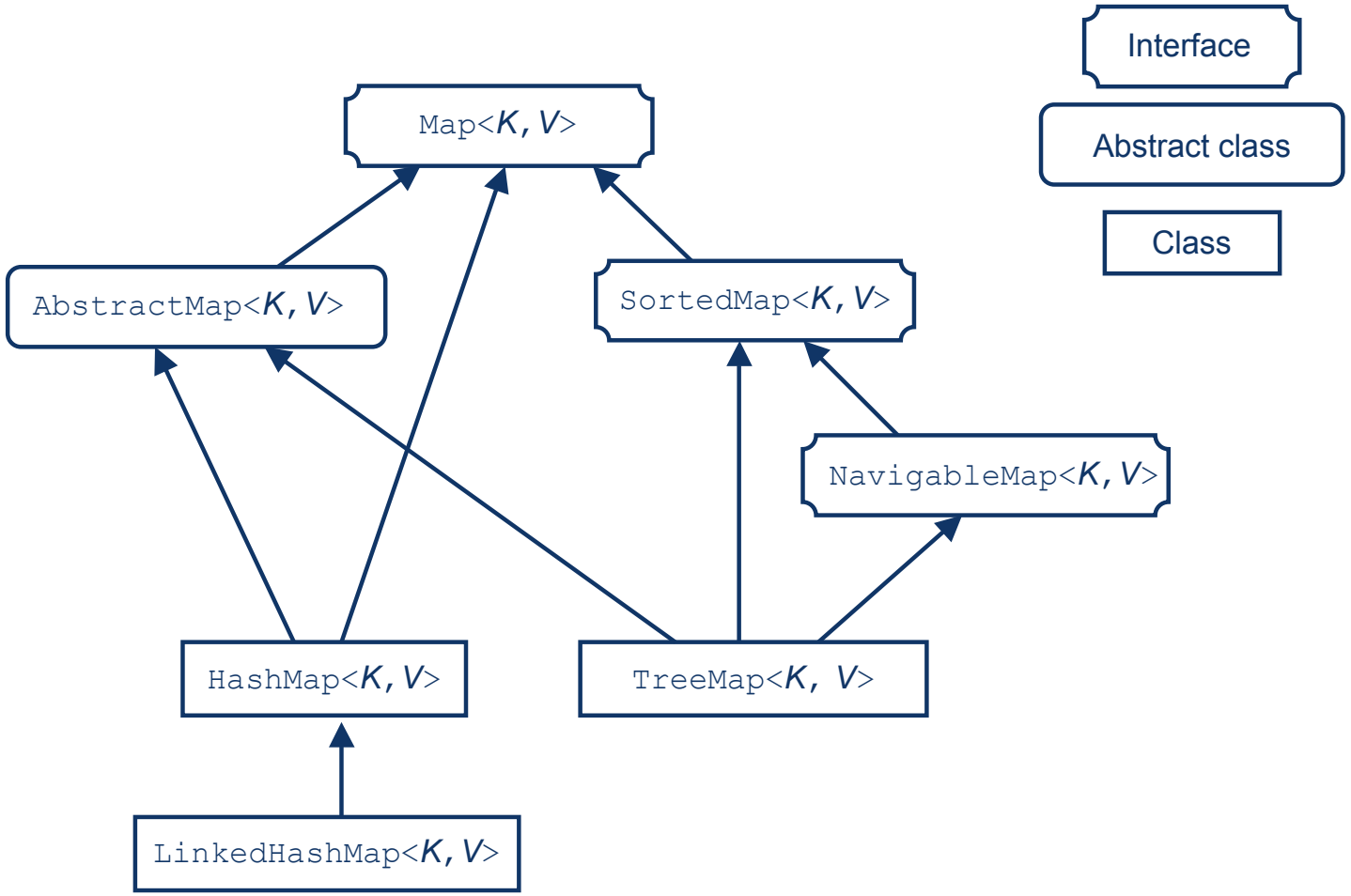
- The concept of a collections framework, recently "invented" by computer programmers, has been known to mathematicians for hundreds of years
- In discrete mathematics complex compound types are methodically constructed from simpler atomic ones:
  - sets
  - tuples (including pairs)
    - relations as sets of pairs
      - functions as special-case relations
        - » multisets as special-case functions
        - » … et cetera

Reference:
E. Billington, D. Donovan, B. Jones, S. Oates-Williams, and A. Street. *Discrete Mathematics: Logic and Structures*. Longman, 1990.

Part of Java's collections framework



Another part of Java's collections framework

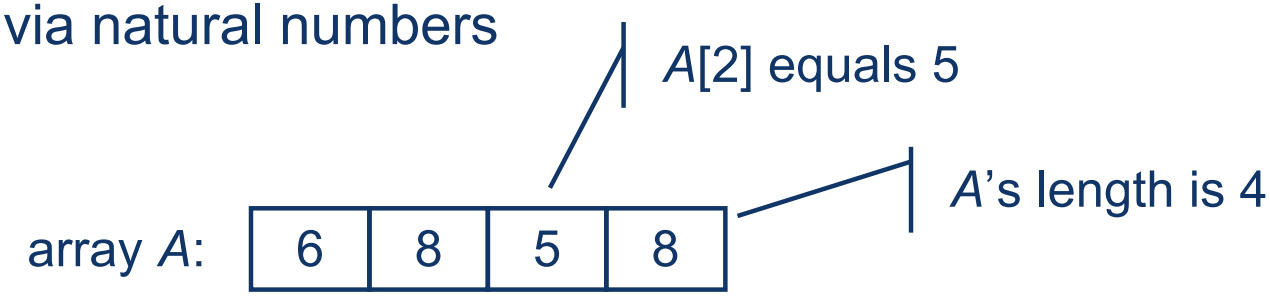


Collections

- A *collection* is a group of *elements* (of the same type)
- Java's `Collection` interface specifies basic operations on groups of elements:
  - `isEmpty()`, `contains()`, `size()`
  - `toArray()`
  - … plus various 'optional' operations

Arrays

- Arrays are:
  - Groups of *fixed size*
  - *Indexable* via natural numbers



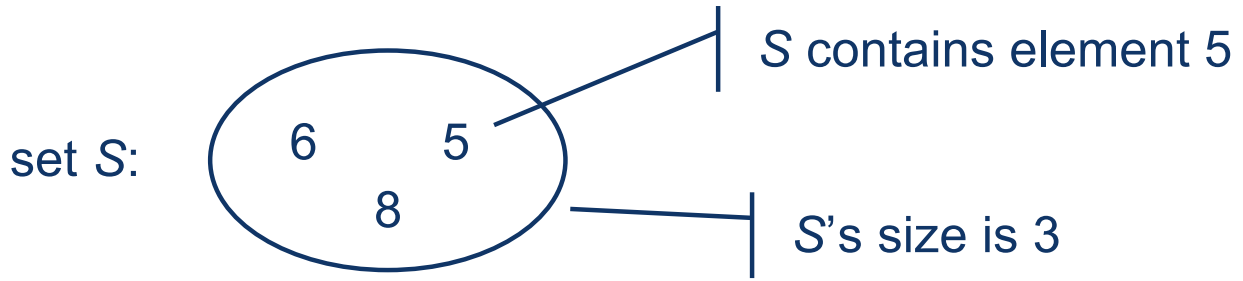
- In Java:
  - For historical reasons, arrays are primitive types, not classes
  - Literal array values can be created via:

```
new int[] {6, 8, 5, 8}
```

  - An array *A*'s length is accessible via *A*`.length`
  - Class `java.util.Arrays` contains various methods for *copying*, *sorting*, *searching* and *range-indexing* arrays

Sets

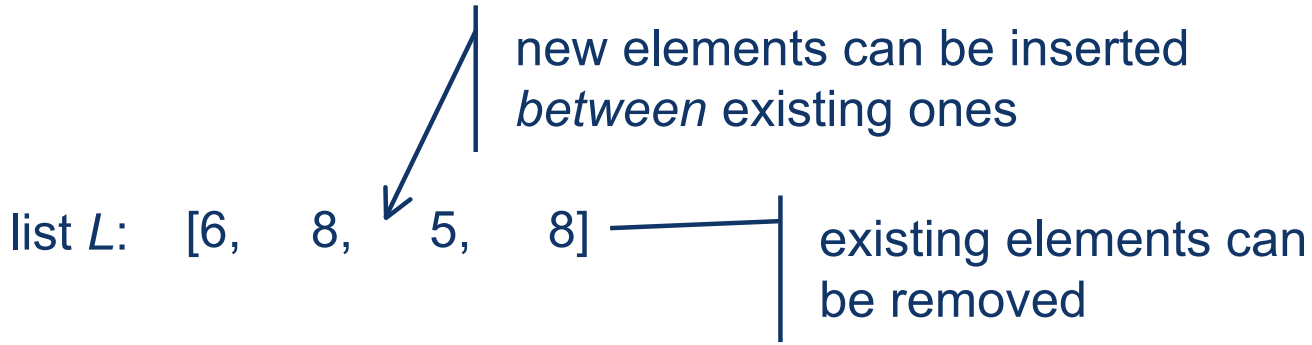
- Sets are:
  - Groups of elements with *no duplicates*
  - Testable for *membership*
  - Are usually *unordered*, but ordered sets are possible



- In Java:
  - Implemented by classes `HashSet` (unordered), `TreeSet` (ordered) and `LinkedHashSet` (insertion-ordered)
  - Methods include `add()`, `remove()`, `isEmpty()`, `size()` and `contains()`

Lists

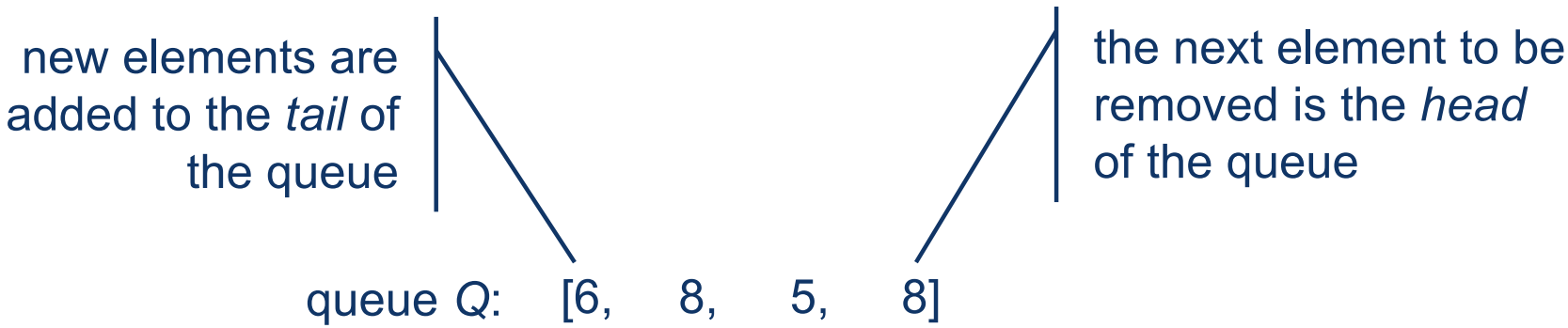
- Lists are:
  - Ordered *sequences* of elements
  - *Indexable* by individual elements or an inclusive-exclusive *range*
  - Can have elements *inserted at* or *removed from* any position



- In Java:
  - Implemented by classes `ArrayList`, `LinkedList`, `Stack` and `Vector`
  - Methods include `get()`, `set()`, `add()`, `remove()`, `indexOf()`, `sort()`, `reverse()`, `rotate()` and `fill()`

Queues

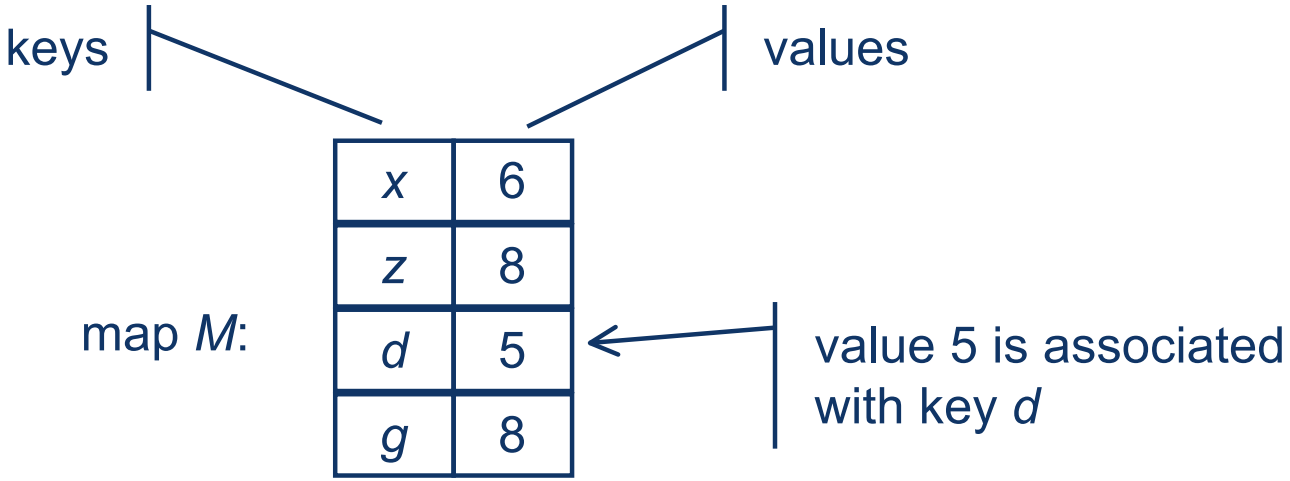
- Queues are:
  - Ordered *sequences* of elements
  - Accessed via their *endpoints* (*head* and *tail*, or *front* and *end*)
  - Structures for FIFO (and LIFO) processing of elements



- In Java:
  - Implemented by classes `ArrayDeque` (double ended), `ArrayBlockingQueue` (fixed capacity) and `PriorityQueue`
  - Methods include `add()`, `peek()` and `remove()`

Maps

- Maps are:
  - Lookup tables that associate (unique) *keys* with *values*



- In Java:
  - Implemented by classes `HashMap` (unordered), `Treemap` (ordered by key) and `LinkedHashMap` (insertion-ordered)
  - Methods include `put()`, `get()`, `containsKey()`, `containsValue()` and `remove()`

Iteration over collections in Java

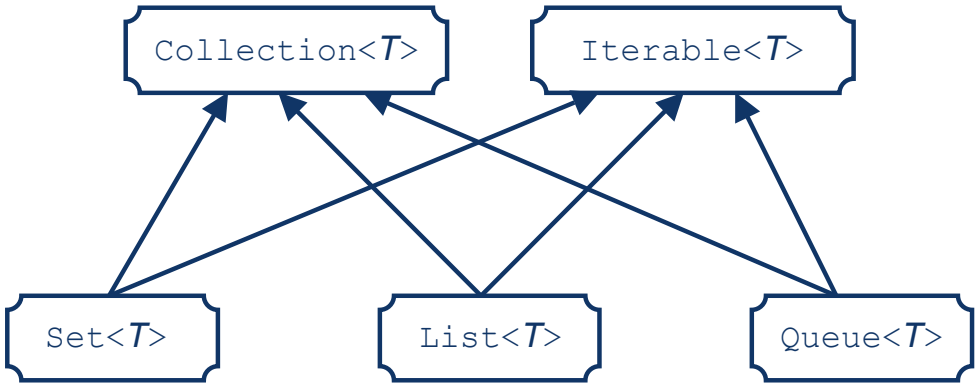
- We're familiar with the idea of using an index variable to iterate over the values in an array, but how can we do this for collections that are not indexable?
- Java's 'for each' style **`for`** loop works for `Collection` types and arrays

```
HashSet<String> names;
… add some names
for (String name : names) {
    System.out.println(name);
}
```

Iteration over collections in Java

- To support this the `Collection` types implement method `iterator()` which returns iteration objects with methods `hasNext()` and `next()`, among others

```
HashSet<String> names;
... add some names
Iterator<String> iter = names.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```



Different implementations of the same collection

- Different implementations are provided for the same collection

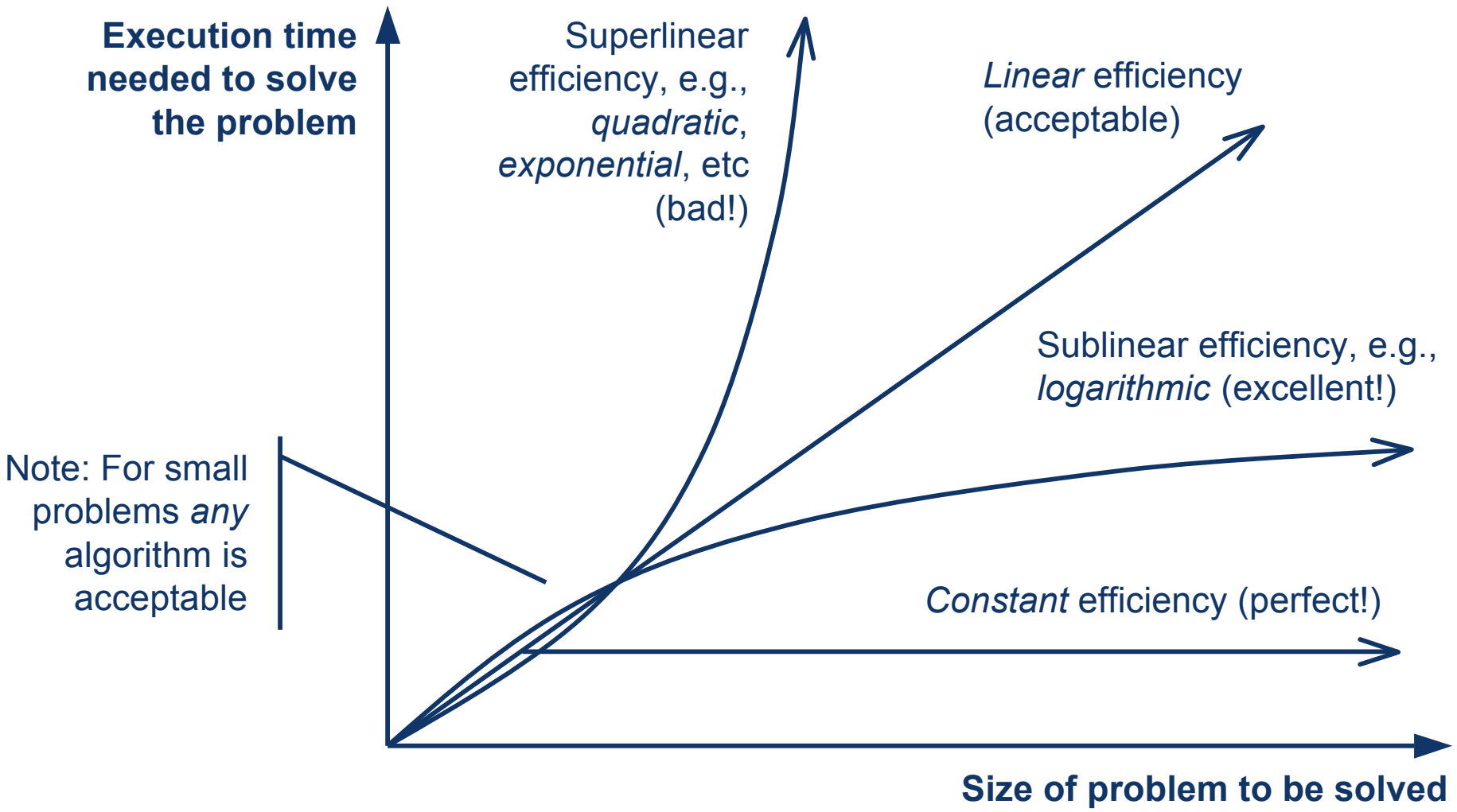
Interface	Implementation: Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table and Linked List
Set	`HashSet`		`TreeSet`		`LinkedHashSet`
List		`ArrayList`		`LinkedList`	
Queue		`ArrayDeque`		`LinkedList`	
Map	`HashMap`		`TreeMap`		`LinkedHashMap`

- This allows us to choose the most appropriate collection implementation for our particular application based on both its functionality and its *efficiency*

Efficiency of pre-defined collections

- An advantage of using pre-defined data types, rather than writing your own, is that they are already optimised for high efficiency
- In general:
  - Fixed-size arrays and hash tables can be indexed in *constant* time
  - Sorted arrays and trees can be searched in *logarithmic* time
  - Linked lists can be accessed in *linear* time

Common efficiency classes



Designing subtypes

- Well-designed subtypes should obey the *substitution principle*: Subtypes should be usable by just following the supertype's specification
  - *Signature rule* - Subtypes must have signature-compatible methods for all of the supertype's methods
  - *Methods rule* - Calls to subtype methods must "behave like" the corresponding supertype methods
  - *Properties rule* - The subtype must preserve all of the provable properties (e.g., invariants) of the supertype

Reference:
B. Liskov and J. Guttag. *Program Development in Java*. Addison-Wesley, 2001.

Part B — Generics

What are generic types?

- Generic types are types that accept another type as a parameter
- All of the classes in the collections framework are generic because the type of their elements is introduced only when a collection is declared
  - Supports compile-time checking of element type homogeneity

Generic types in Java

- Classes (and methods) can have generic type parameters, denoted <*X*>
- *Bounded* generic types, which restrict the actual type parameter allowed, can be specified by saying which class the allowed types **`extend`**
- Although an object of subtype *B* can be used where an object of its supertype *A* is expected, generic type *G*<*B*> is *not* a subtype of *G*<*A*>

Part C — Exceptions

What are exceptions?

- Exceptions are undesirable events that are outside the “normal” behaviour of a program
  - It may be possible to *recover* from an exception
  - Exceptions can be *handled* in the method where they occur or can be *propagated* to the calling program
  - Intended behaviours should not be coded using exceptions
- Advantages:
  - Error handling is separated from normal code
  - Allows different kinds of errors to be distinguished

Exceptions in Java

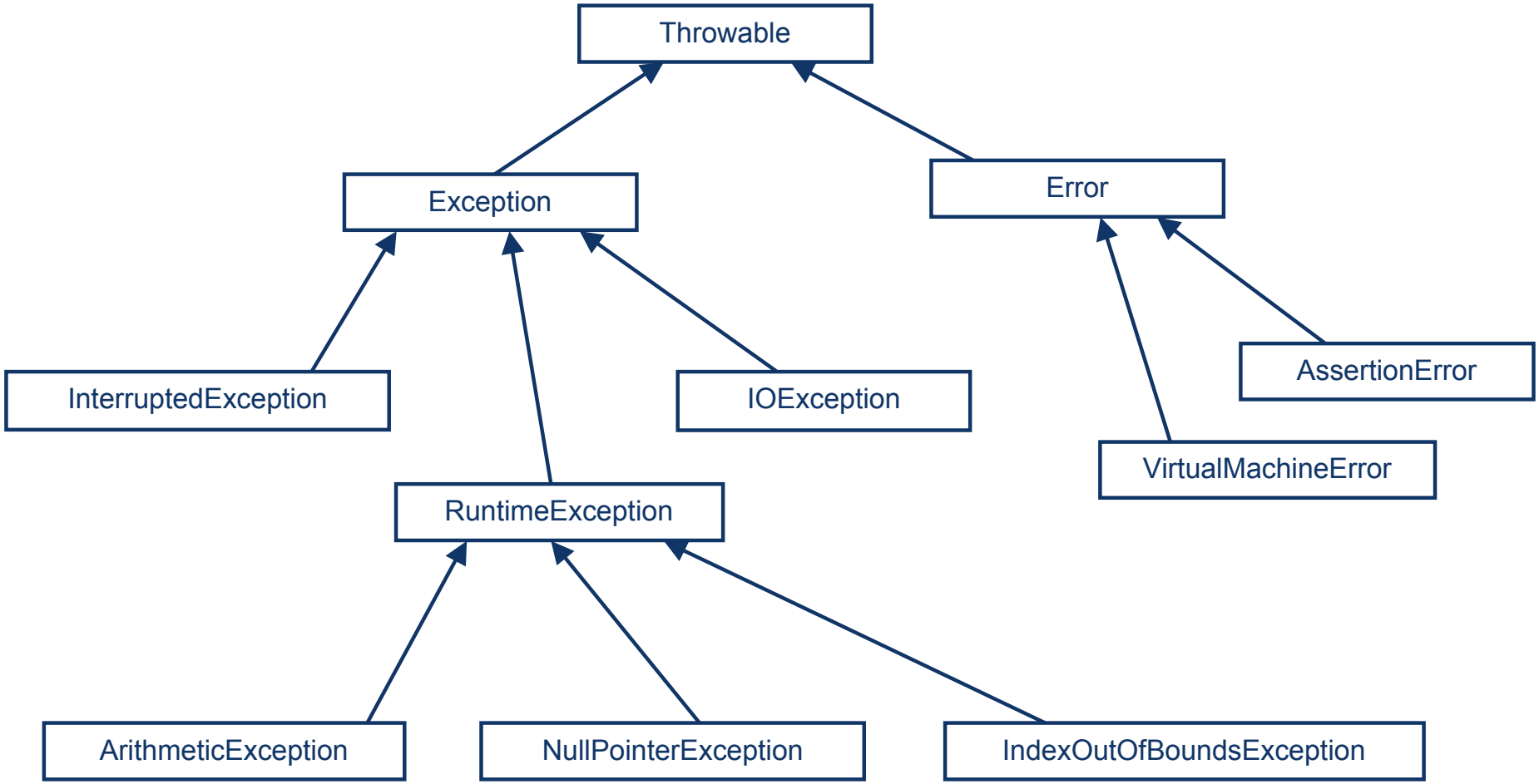
- Exception objects are *thrown* by code that *tries* to run but fails, and are *caught* by an exception handler:

  `try` `{`… *normal code*`}`

  `catch``(`*exception-class object*`) {`… *exception-handling code*`}`
- Catching an exception also catches its subtype exceptions
- Failure of *all* program units to catch an exception results in the program terminating with a run-time error
- Exceptions are a class and form a type hierarchy
  - You can define your own exceptions by subtyping class `Throwable`
  - You can deliberately throw an exception if some problem is identified:

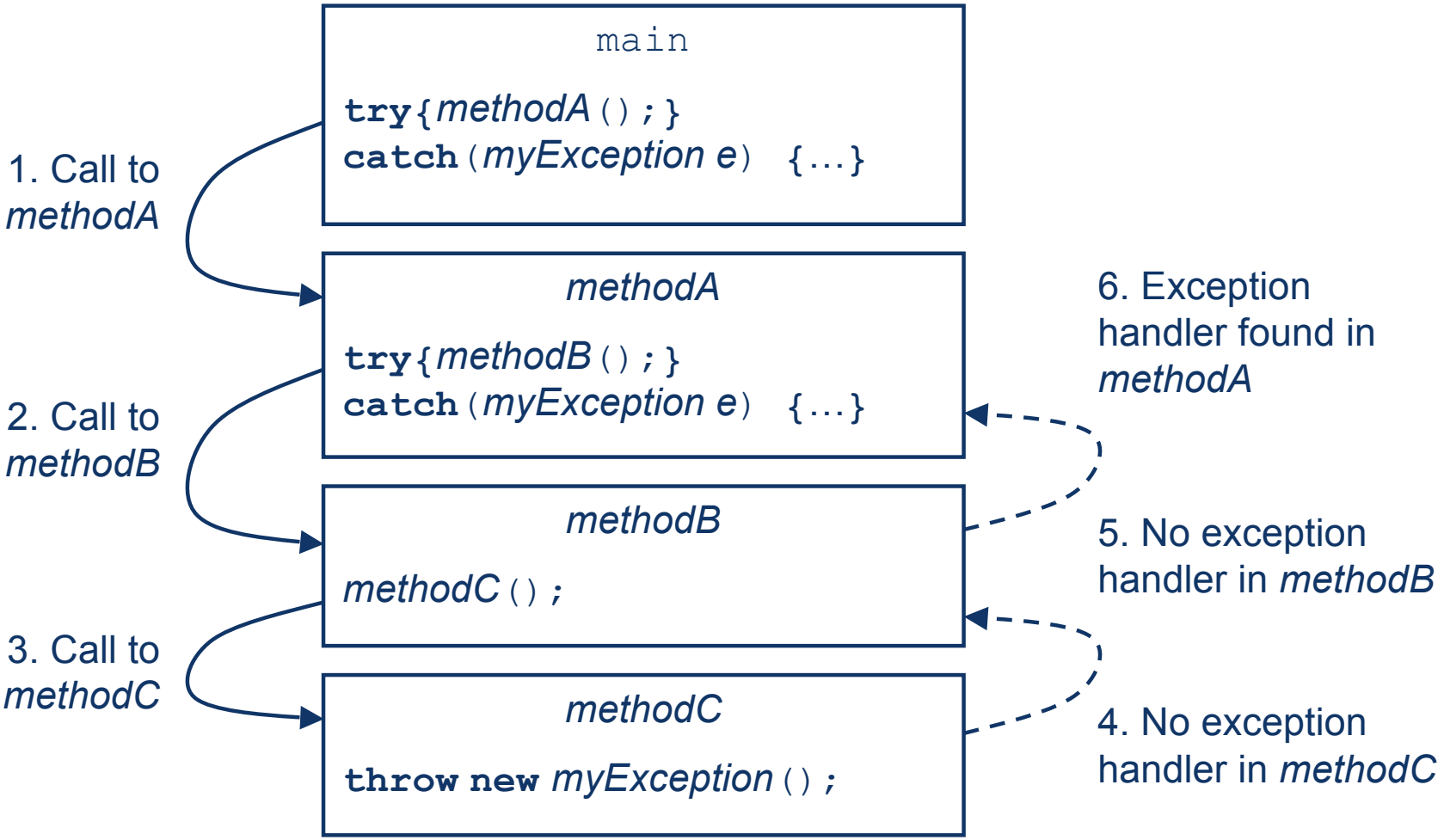
    `throw` *exception-object*`;`

Some of Java's exception types



Reference:
B. Holmes. *Programming with Java*. Jones and Bartlett, 1998.

Propagation of exceptions



Reference:
B. Holmes. *Programming with Java*. Jones and Bartlett, 1998.

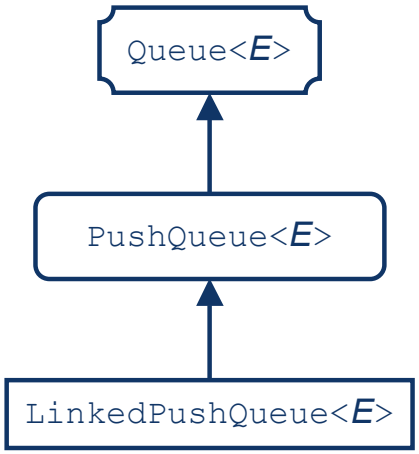
Exceptions in Java

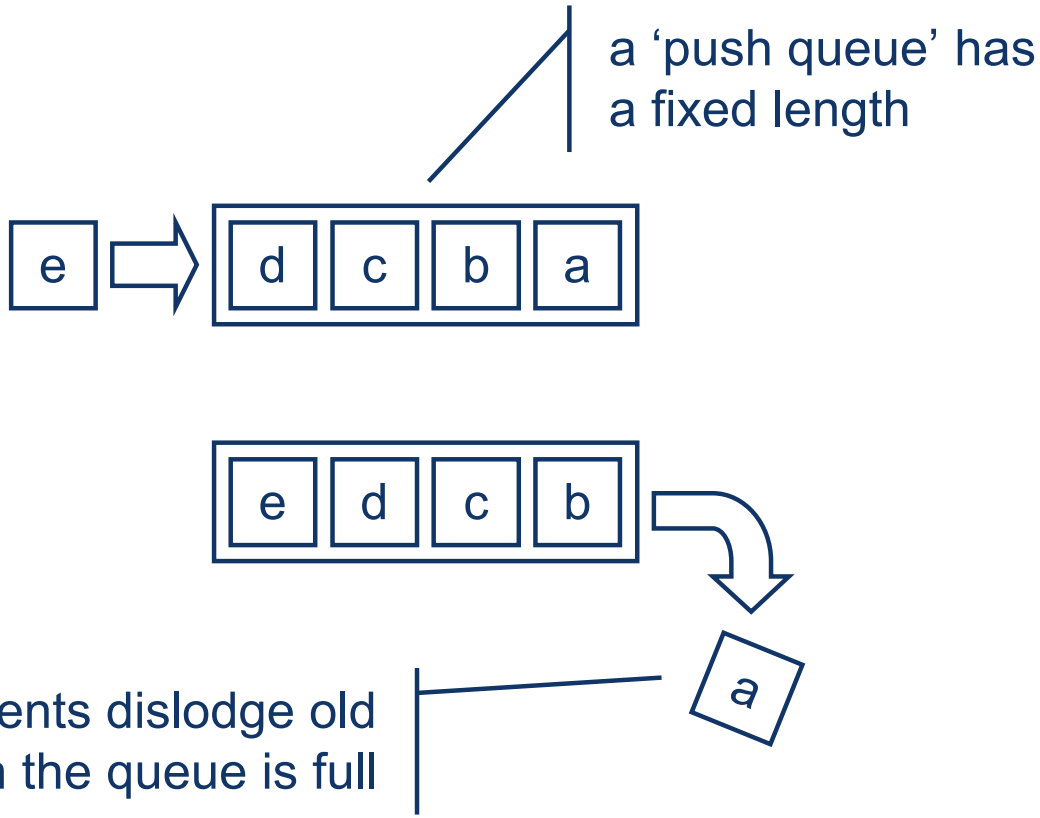
- At compile-time a method that *may* throw a 'checked' exception must either:
  - Include exception-handling code to **`catch`** the exception; or
  - Declare to its callers that it potentially **`throws`** the exception
- Checked exceptions are those that a well-written program should reasonably be expected to handle
- 'Unchecked' exceptions, comprising *errors* and *runtime exceptions*, are not bound by this requirement
  - Unchecked exceptions are the result of *programming errors*, and are not part of the deliberate API

Part D — Demonstration

Demonstration

- Implementation of a generic class `PushQueue`, based on the existing collections framework class `Queue`





Homework

- Read Sun's Java Tutorial on the Collections Interface

  `http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html`

- Read Sun's Java Tutorial on Generics

  `http://java.sun.com/docs/books/tutorial/java/generics`

- Read Sun's Java Tutorial on Exceptions

  `http://java.sun.com/docs/books/tutorial/essential/exceptions`