Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

Input / Output

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS2018

Computer science

Motivation

- Saving, transferring and reconstructing state information of a Java application.
  - E.g. session state information of web browser, i.e. list of currently displayed URLs.

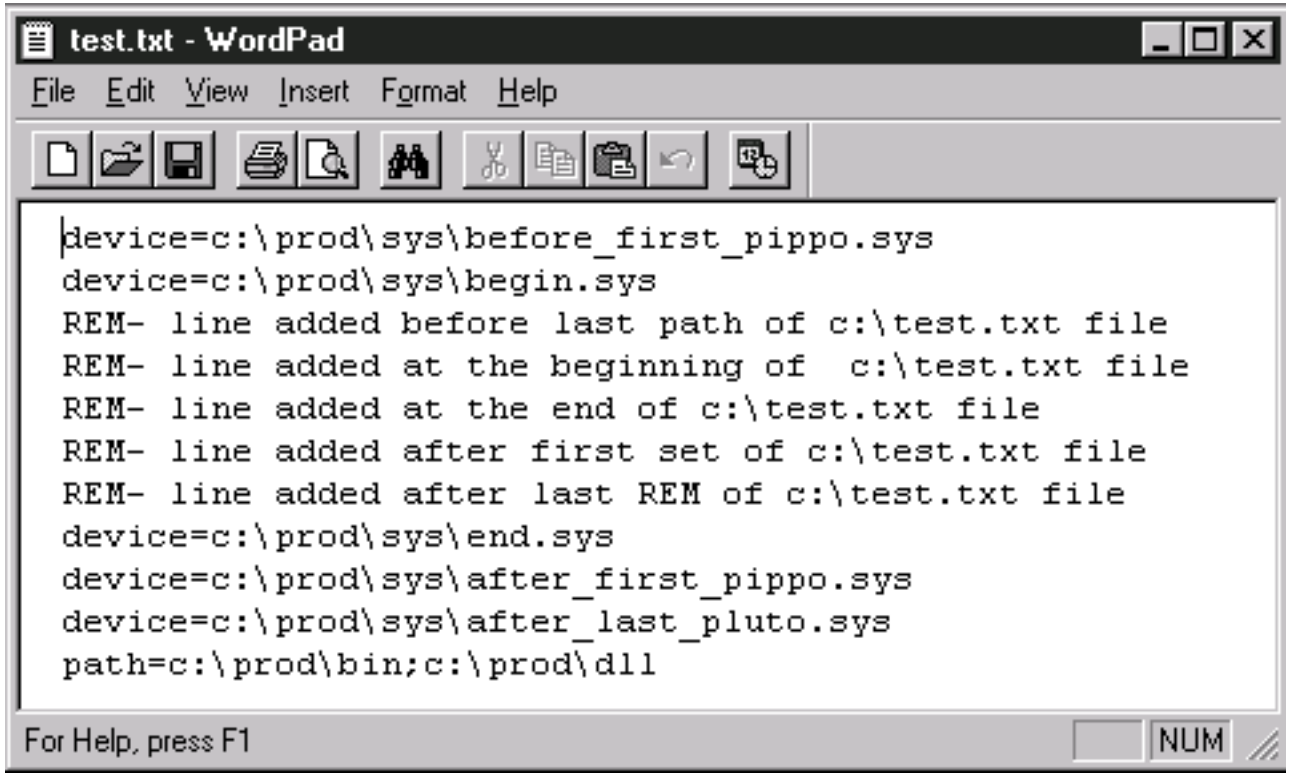
Outline – Input / Output

- Text Files
- Binary Files
- Random File Access
- Serialization

Text Files

(Plain) Text files contain a sequence of ordinary textual characters.

- Human readable
- Examples: txt, xml, csv, log, Java source code …



Outline – Input / Output

- Text Files
- Binary Files
- Random File Access
- Serialization

Processing I/O with Text Files

- Text file contains a sequence of characters
  - Integer 12345 stored as characters '1' '2' '3' '4' '5‘
- Use `Reader` and `Writer` and their subclasses to process I/O with text files.
- To read:

```
FileReader reader = new FileReader("input.txt");
```

- To write:

```
PrintWriter out =    new PrintWriter("output.txt");
```

or

```
FileWriter out2 = new FileWriter("output.txt");
```

Path names

- A **file name** can contain the **path** and a **drive**.
- A path instance reflects the underlying platform:
  - Linux/Mac: /home/joe/foo
  - Windows: C:\home\joe\foo

- Representation as a **`String`**

```
FileReader in = new FileReader("c:\\homework\\input.dat");
```

- **Recall**: a single backslash inside quoted strings is an escape character that is combined with another character to form a special meaning

Path interface in Java 7/8 (1/2)

- NIO (New Input Output) API
- A ***Path*** is a sequence of directory names, optionally followed by a file name.
  - First component may by a root component, e.g. / or *C:*
  - A path starting with a root component is *absolute*, otherwise *relative*.

```
Path absolute = Paths.get("C:","local_data","Prog2_FS18");
Path relative = Paths.get("temp","conf","user.properties");
```

  - The strings are joined with the separator of the file system (/ for Unix/Mac, \ for Windows)

Path interface in Java 7/8 (2/2)

- ***Paths.get() method***
  - receives one or more strings
  - Joins the strings with the separator of the file system (/ for unix/mac, \ for Windows)
  - Validates if the paths exists and throws an `InvalidPathException`, if not

```
Path absolute = Paths.get("C:","local_data","Prog2_FS15");
Path relative = Paths.get("temp","conf","user.properties");
Path homedir = Paths("/home/lua1");
```

Reading Text Files

- To read from a file, construct a `FileReader`
- Use the `FileReader` to construct a `Scanner` object

```
FileReader reader = new FileReader("input.txt");
Scanner scanner = new Scanner(reader);
```

- Use the `Scanner` methods to read and parse data from file
  - `nextLine()`, `nextInt()`, `nextDouble()`, `next()`, …

- Important: call `reader.close()` and `scanner.close()` to close the file reader and the scanner. If you don't your program will introduce a resource leak.

Writing Text Files

- To write to a file, construct a **`PrintWriter`** or **`FileWriter`** object

```
PrintWriter out = new PrintWriter("output.txt");
FileWriter out2 = new FileWriter("output.txt");
```

  - If the file exists, it is overwritten.
  - If file doesn't exist, an empty file is created.
- Use **`print()`**, **`printf()`** and **`println()`** to write into a **`PrintWriter`** (more powerful)

```
out.println(29.95); out.println("Hello, World!");
```

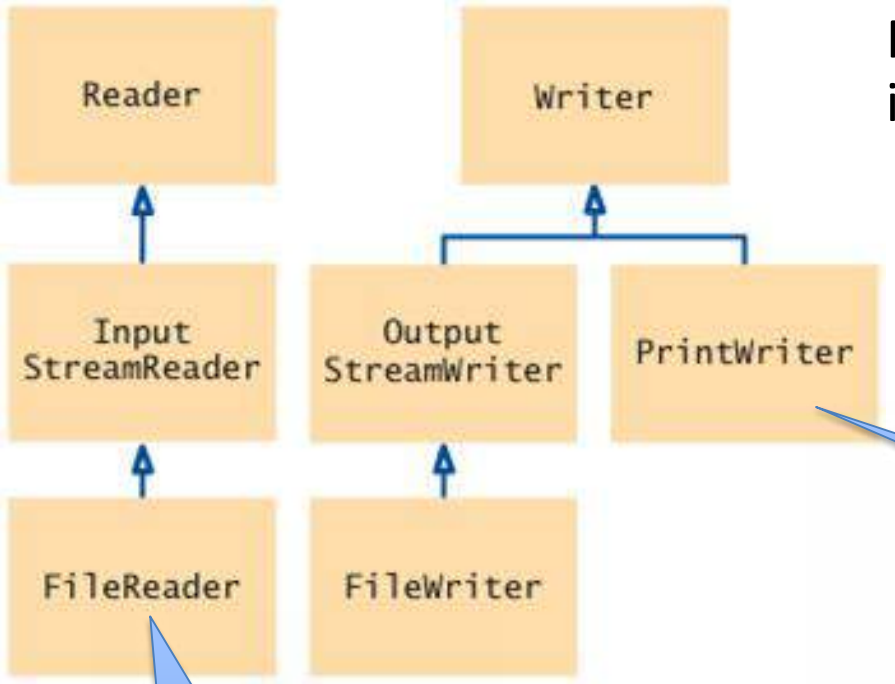
- Use **`write()`** to write into a **`FileWriter`**:

```
out2.write("Hello, World!");
```

- Close the file after processing to avoid resource leak. Otherwise, output may not be completely written to the file.

```
out.close(); out2.close();
```

Reading/writing text files



Readers and writers handle data in text form

Works like **`System.out`**

Can be used together with **`Scanner`**

NIO Text Files

- Read a file as a sequence of lines

```
List<String> lines = Files.readAllLines(path);
```

- Write a collection of lines into a file

```
Files.write(path, lines);
```

- 2 lines replace a lot of code !!!

Example: Line by Line Processing

- Application for numbering files, e.g. files containing Java source code.
- Reads all lines of a file and sends them to an output file, preceded by line numbers.
- Sample input file:

```
Mary had a little lam
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

- Output file:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

Reading a Single Character

- Use `read()` method of `Reader` class to read a single character
  - returns the next character as an as integer or the integer -1 at end of file.

- Example

```
Reader reader = ...;
int next = reader.read();
char c;
if (next != -1) {
   c = (char) next;
   // …
}
```

- Hint: class `FileReader` is a subclass of `Reader`

Processing Text Input

1. String Processing
   - Use the methods of the `scanner` class
     - nextLine(), nextInt(), nextDouble(),…
   - Use methods `Character.isDigit()` and `Character.isWhiteSpace()` to further process strings

2. Regular Expressions
   - Define a delimiter `("\t")` and split the string there

```
String[] items = line.split("\t");
```

Regular Expressions

- **Regular expressions** are a way to describe a **set of strings** based on common characteristics shared by each string in the set.
- used to search, edit, or manipulate text and data.

- Composed of
  - String literals, e.g. `abc`
  - Meta characters: `<([{\^-=$!|]})?*+.>`

- Example usage in: `RegexTestHarness.java`

Regular Expressions

- Character classes

Construct	Description	Examples
[abc]	a, b, or c (simple class)	[bcr]at → bat, cat, rat
[^abc]	Any character except a, b, or c (negation)	[^bcr]at → hat, …
[a-zA-Z]	a through z, or A through Z, inclusive (range)	foo[1-5] → foo1, … foo5
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)	[0-2[6-8]] → 0,1,2,6,7,8
[a-z&&[def]]	d, e, or f (intersection)	[2-8&&[4-6]] → 4,5,6
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)	[0-6&&[^345]] → 0,1,2,6
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)	[0-6&&[^3-5]] → 0,1,2,6

Regular Expressions

- Predefined Character classes

Construct	Description
.	Any character
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word characters: [^a-zA-Z_0-9]

Regular Expressions

- Quantifiers

Construct	Meaning	Example
X?	X, once or not at all	(ab)? → ab
X*	X, zero or more times	a* → «», a, aa, aaa, …
X+	X, one or more times	a+ →a, aa.¨, aaa, …
X{n}	X, exactly n times	a{3} → aaa
X{n,}	X, at least n times	a{2,} → aa, aaa,,aaaa,…
X{n,m}	X, at least n but not more than m times	a{2,4} → aa, aaa,,aaaa

- Quantifiers can be greedy (X?), reluctant (X??), and possessive (X?+)

Regular Expressions - Exercise

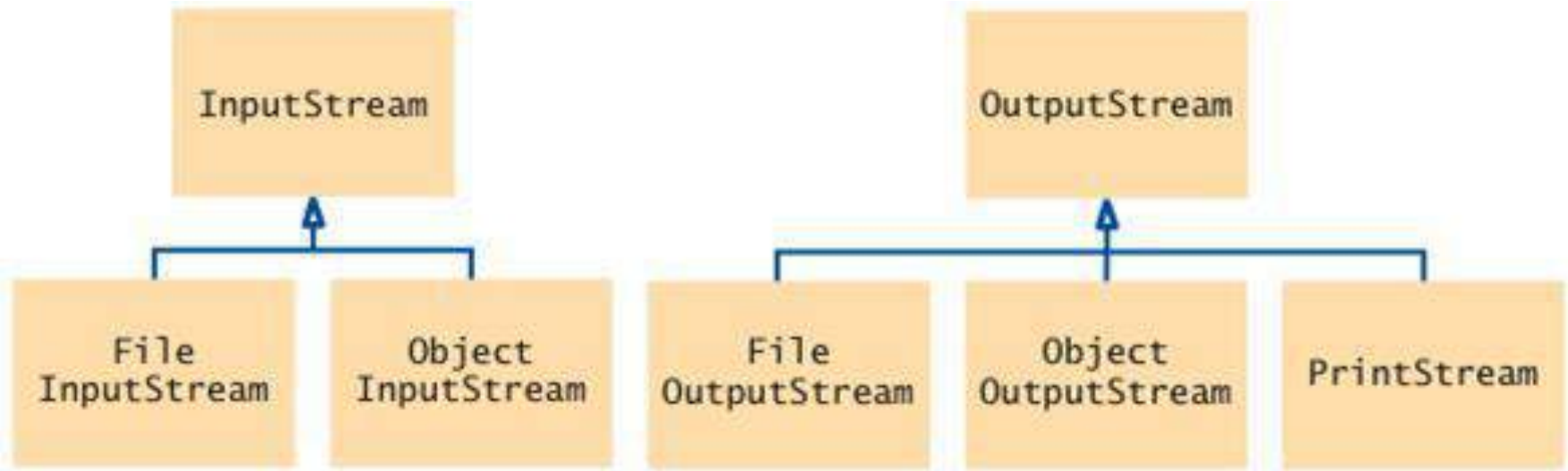
Reg Exp	String	Entire match	Partly match
ab+	aabb		
(ab)+	abab		
(ab)?	abab		
\w{4}	abcd		
\w{4}	abcdef		
[3-5]*	4444		
foo[0-9&&[^13579]]	foo23		
[2-4]{1,3}	23456		
[1-9]+[0-9]*	060742		

Outline – Input / Output

- Text Files
- Binary Files
- Random File Access
- Serialization

Processing I/O with Binary Files

- data is stored byte by byte
- Ex: Integer 12’345 stored as a byte sequence: 00 00 48 57
- Use `InputStream` and `OutputStream` and subclasses



Processing I/O with Binary Files

- To read:

```
InputStream inputStream = new FileInputStream("input.bin");
```

- To write:

```
OutputStream outputStream = new FileOutputStream("output.bin");
```

Reading a Single Byte

- Use `read` method of `InputStream` class to read a single byte:
  - Returns the next byte as int or the integer -1 at end of file

- Example

```
InputStream in = ...;
int next = in.read();
byte b;
if (next != -1) b = (byte) next;
```

NIO Binary Files

- To read:

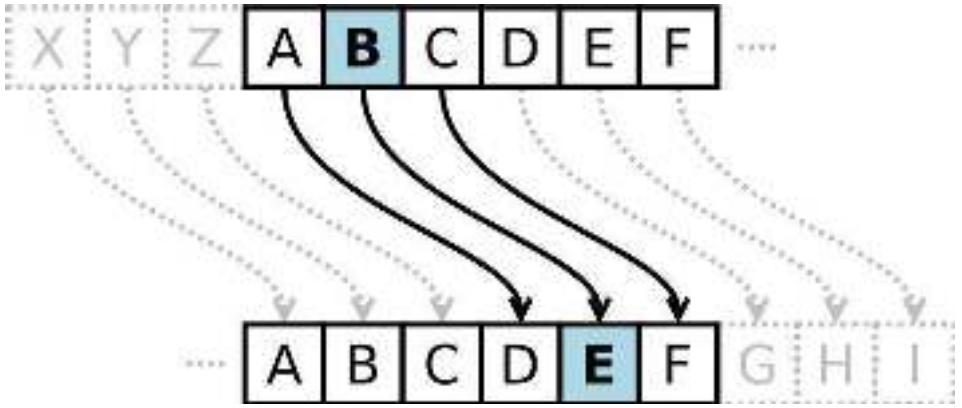
```
InputStream inputStream = Files.newInputStream(Paths.get(inFile));
```

- To write:

```
OutputStream outputStream =   Files.newOutputStream(Paths.get(outFile));
```

- To read/write a whole file:
- `byte[] bytes = Files.readAllBytes(path);`
- `Files.write(path, bytes);`

Example: Byte by Byte Encryption



- **Caesar cipher**
  - Each letter is replaced by a letter some fixednumber of positions further down the alphabet.
  - The fixed number is called the “encryption key”.
  - To decrypt, use the negative of the encryption key

- **Warning:** This cipher can easily be broken. Our focus is on byte processing and not on security.

Command line arguments

- Passing parameters at start
- Example: `java LineNumberer input.txt numbered.txt`

- Use the `args` parameter of the `main` method

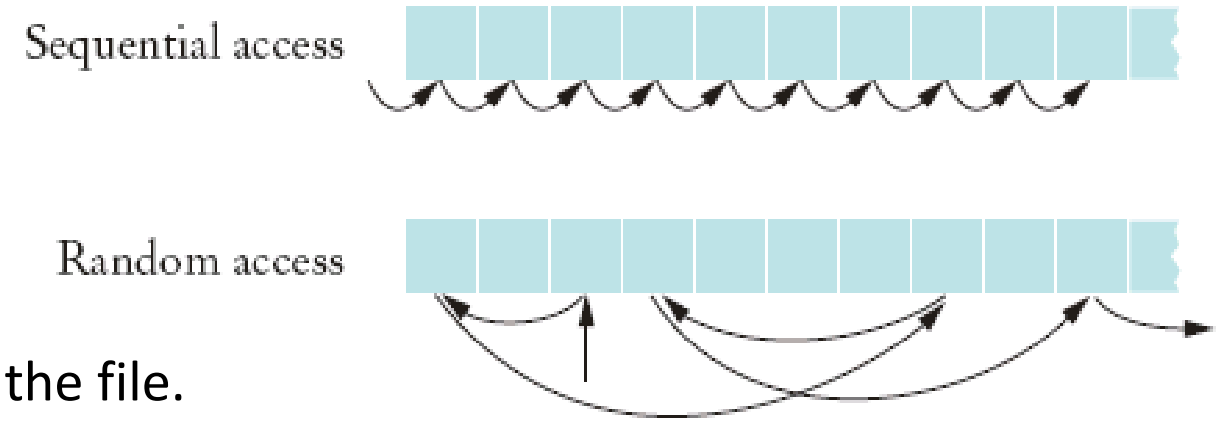
```
public class Echo {
   public static void main (String[] args) {
    for (String s: args) {
       System.out.println(s);
    }
   }
}
```

Outline – Input / Output

- Text Files
- Binary Files
- Random File Access (RAF)
- Serialization

Random File Access

- **Sequential file access**
  - A file is processed line by line, byte by byte, etc.
  - Sequential processing can be very inefficient.
  - E.g. retrieving a record in a database.



- **Random file access (RAF)**
  - Allows access at arbitrary locations in the file.
  - Files support random access; System.in and System.out do not
  - Each file has a special file pointer position. You can read or write at the position where the pointer is.

Random File Access - Records

- Structured data is written in records
- A record is a collection of *elements*, possibly of different data types, typically in fixed number , length and sequence.

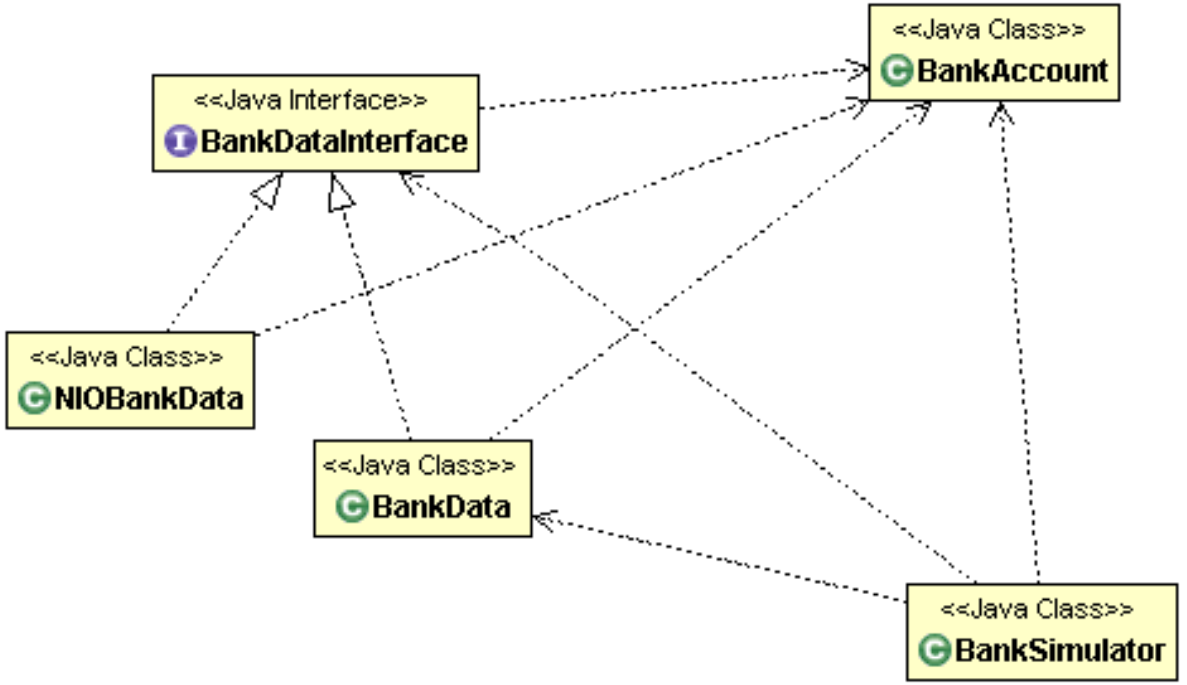
- Used in databases: each line in a table is a record.

Random File Access

Java	< 7	
Interface	–	`SeekableByteChannel`
Class	`RandomAccessFile`	`FileChannel`
Position the pointer	`seek(long n)`	`position(long n)`
File pointer current position	`getFilePointer()`	`position()`
Get current file size (in byte)	`length()`	`size()`
read	`read…() for all datatypes`	`read(ByteBuffer b)`
write	`write…(…) for all datatypes`	`write(ByteBuffer b)`
close	`close()`	`close()`

Exercise: Store and Retrieve Bank Accounts

- Application that uses a random access file to store records with bank account data
- Program lets you pick a record and deposit money into this account
- Please have a special look at the formats employed in the code



Outline – Input / Output

- Text Files
- Binary Files
- Random File Access (RAF)
- Serialization

Serialization

- Standard method to read and write objects
- Use **Streams**:
  - `ObjectOutputStream` to write
  - `ObjectInputStream` to read
- These classes can save and read streams of objects.

Serialization

- **Serialization**: process of saving objects to a stream
  - Each object is assigned a serial number on the stream
  - If the same object is saved twice, only serial number is written out the second time.
  - When reading, duplicate serial numbers are restored as references to the same object.
- Objects that are written to an object stream must belong to a class that implements the `Serializable` interface.
  - `Serializable` interface has no methods.

Example I

- Writing a `BankAccount` object to a file

```
class BankAccount implements Serializable {
  ...
}

BankAccount b = ...;

ObjectOutputStream out = new ObjectOutputStream
     (new FileOutputStream("bank.dat"));

out.writeObject(b);
```

- The object output stream takes care of saving the bank account including its balance

Example I

- Reading a `BankAccount` object from a file.

```
ObjectInputStream in = new ObjectInputStream
     (new FileInputStream("bank.dat"));

BankAccount b = (BankAccount) in.readObject();
```

- The `readObject` method
  - Returns an object reference; this reference needs to be casted to the original type
  - Can throw `ClassNotFoundException`
    - checked exception → catch or declare.

Example II

- Read and write an `arrayList` of bank accounts to a file.
- Write

```
ArrayList<BankAccount> a = new ArrayList<BankAccount>();
// Now add BankAccount objects into this arrayList
out.writeObject(a);
```

- Read

```
ArrayList<BankAccount> a = (ArrayList<BankAccount>) in.readObject();
```

- The object output stream takes care of saving the array list including **all** bank accounts and their balances.

Example III

- Application that reads and writes a `Bank` object containing an `arrayList` of `BankAccounts`.
- The object output stream takes care of saving
  - The bank object
  - The array list of bank accounts in the bank object
  - The bank accounts in the array list
  - The balances in the bank accounts.

Deep cloning using serialization

The steps for making a deep copy using serialization are:

1. Ensure that all classes in the object's graph are serializable.
2. Create input and output streams.
3. Use the input and output streams to create object input and object output streams.
4. Pass the object that you want to copy to the object output stream.
5. Read the new object from the object input stream and cast it back to the class of the object you sent.

Deep cloning using serialization

- Demo: ObjectCloner.java

- Runtime speed comparison (Milliseconds to deep copy a simple class graph n times)

Iterations	1000	10000	100000
clone()	10	101	791
Serialization	1832	11346	107725

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

Exceptions

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS2018

Motivation

Explosion of the Ariane 5 1996

http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

Outline - Exceptions

- What are exceptions?
- Throwing exceptions
- Exception Handling
- Finally Clause
- Self-made Exception Types

Outline - Exceptions

- What are exceptions?
- Throwing exceptions
- Exception Handling
- Finally Clause
- Self-made Exception Types

Review questions

1. What is an exception?
2. Why do I need exceptions?
3. Which exceptions should be handled?
4. What is the difference between checked and unchecked exceptions?
5. Why it is important to know the exception class hierarchy?

What is an Exception?

- "Exceptional event"

- **Definition**:

  An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program.

- When an error occurs within a method, the method creates an object and hands it off to the runtime system.
- This exception object contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called throwing an exception.

Outline - Exceptions

- What are exceptions?
- Throwing exceptions
- Exception Handling
- Finally Clause
- Self-made Exception Types

Throw Exception Example

Method throws exception, e.g.

```
public class BankAccount {
…
   public void withdraw(double amount) {
      if (amount > balance) {
         throw new IllegalArgumentException(
            "Amount exceeds balance");
      }
      balance = balance - amount;
   }
…
}
```

Not executed, when exception is thrown

Throwing an Exception

- Example: argument has an illegal value

```
IllegalArgumentException exception =
new IllegalArgumentException(
    "Amount exceeds balance");
throw exception;
```

- No need to store exception object in a variable:

```
throw new IllegalArgumentException(
   "Amount exceeds balance");
```

- When an exception is thrown, the program flow in a method terminates immediately.

  The program flow resumes in an exception handler.

Error Handling in Java

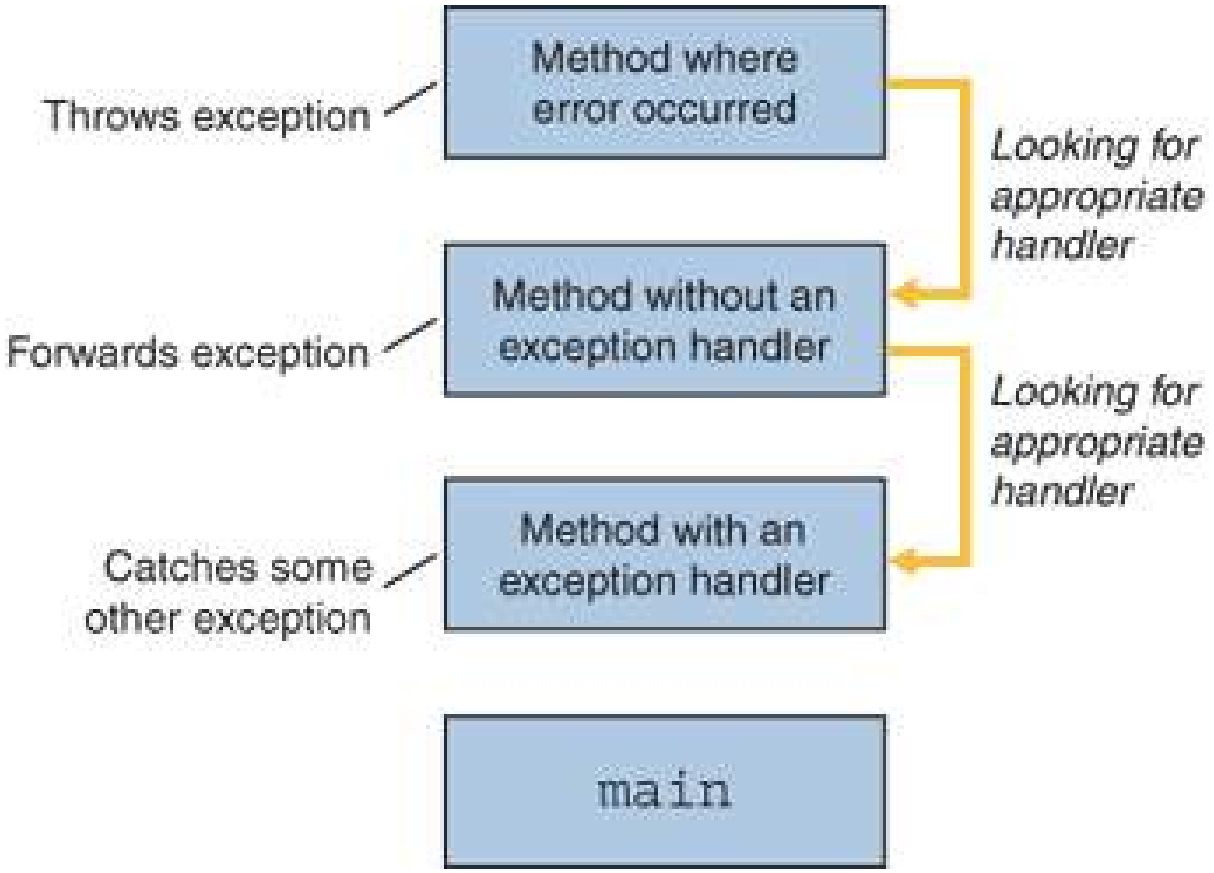


Fig.: Call Stack (upside down)

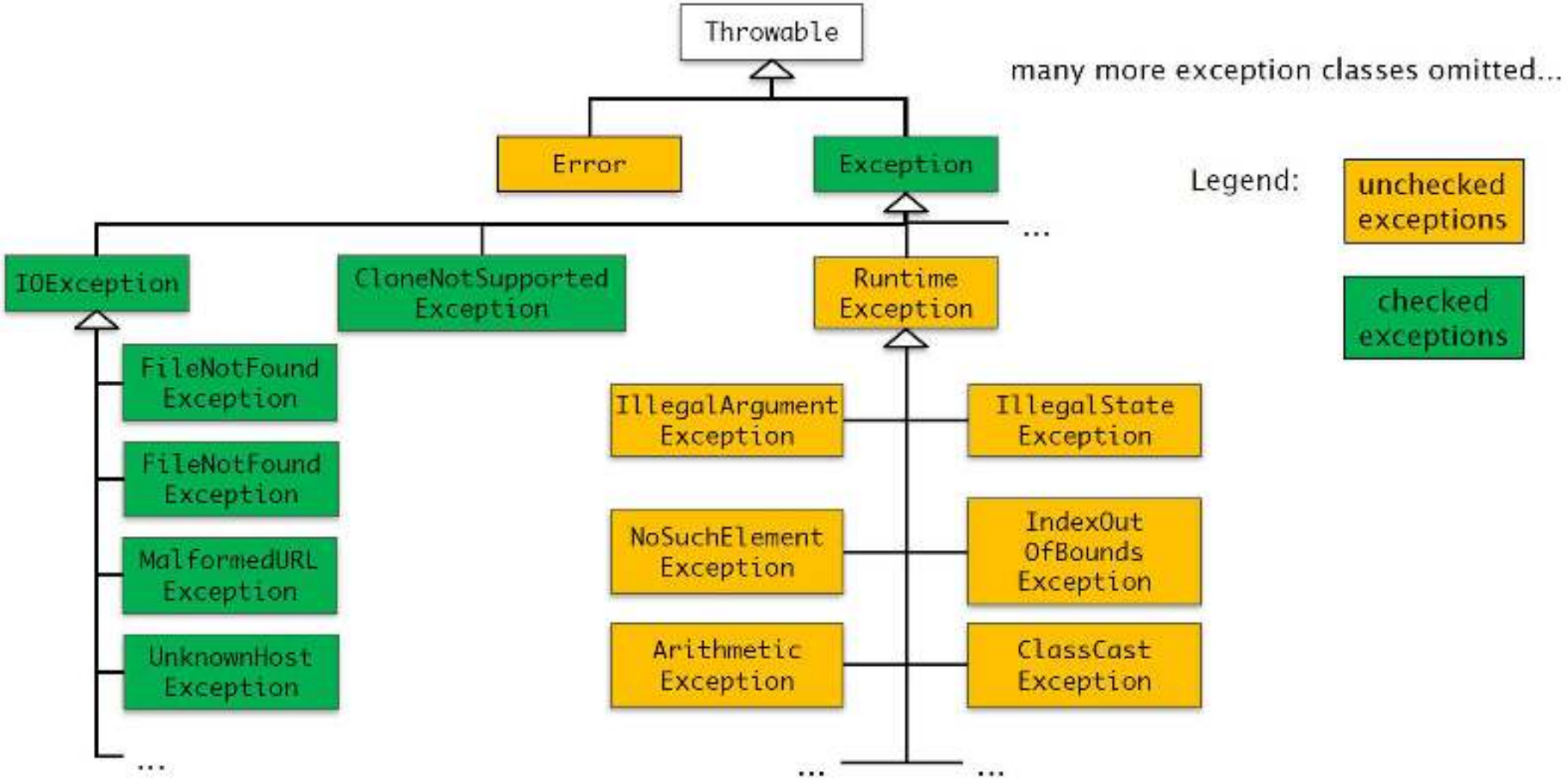
Source: java.sun.com

- Throw an exception if normal program flow is disrupted.
- Forward the exception upward the hierarchy in the call stack.
- Deal with the exception at a level where you can remedy the situation.
  - Start a dialog with the user, etc.

Two Kinds of Exceptions

- **Checked exceptions**
  - Events that a well-written application should anticipate and recover from.
    - E.g. user supplies the name of a nonexistent file, and the constructor throws `java.io.FileNotFoundException`
- **Unchecked exceptions**
  - **Errors**
    - Events that are external to the application, and that the application usually cannot anticipate or recover from.
    - E.g., application is unable to read a file because of a hardware or system malfunction.
  - **Runtime exceptions**
    - Events that are internal to the application, and that the application usually cannot anticipate or recover from.
    - E.g., if a logic error causes a null to be passed to the constructor, the constructor throws a `NullPointerException.`

Hierarchy of Exception Classes



Exception Handling and I/O

- Note that the use of exceptions is not always straightforward.
  - `Scanner.nextInt` throws unchecked `InputMismatchException`
  - Programmer cannot prevent users from entering incorrect input
  - The reason for this choice is to make the class easy to use for beginning programmers.

Outline - Exceptions

- What are exceptions?
- Throwing exceptions
- Exception Handling
- Finally Clause
- Self-made Exception Types

Exception Handling and I/O

- Rule of thumb: deal with checked exceptions primarily when programming with files and streams.

- Deal with exceptions when employing a Scanner to read a file

```
String filename = . . .;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
```

- Deal with the fact that the `FileReader` constructor can throw a `FileNotFoundException`.

Dealing with Exceptions 1/2

Two choices:

1. **Handle the exception locally**
   - Use try/catch blocks

Dealing with Exceptions 2/2

2. **Handle the exception further up the in call stack hierarchy**
   - Use **throws** specifier in the method declaration

```
public void read(String filename) throws FileNotFoundException {
 FileReader reader = new FileReader(filename);
 Scanner in = new Scanner(reader);
 ...
}
```

   - The method could throw multiple types of exceptions

```
public void read(String filename) throws IOException, ClassNotFoundException
```

   - Keep in mind inheritance hierarchy:
     If method can throw an IOException and FileNotFoundException, only use IOException

Rule of thumb: if you need more knowledge about the context, forward the exception up the call stack hierarchy.

Programming with Exceptions

- Execute statements that may cause an exception in the try block
- Provide catch clause for each type of exception and handle the exception

- Example:

```
try {
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
} catch (IOException exception) {
   exception.printStackTrace();
} catch (NumberFormatException exception) {
   System.out.println("Input was not a number");
}
```

- `NoSuchElementException` remains thrown until caught by a surrounding try block.

Programming with Exceptions

- Execute statements in try block.

- If no exceptions occur, catch clauses are skipped.
- If an **exception** of a matching type occurs, execution jumps to catch clause.
- If an exception of another type occurs, this exception remains thrown until it is caught by a surrounding try block (possibly further up in the call stack).

- catch (IOException exception) block
  - exception contains reference to the exception object that was thrown.
  - catch clause can analyze object to find out more details.
  - `exception.printStackTrace()` :
    printout of chain of method calls that lead to exception.

Advantages of Try/Catch

- Separating error-handling code from “regular" code.
- Propagating errors up the call Stack to have more context
- Grouping and differentiating error types.
- For details see http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html

Outline - Exceptions

- What are exceptions?
- Throwing exceptions
- Exception Handling
- Finally Clause
- Self-made Exception Types

Finally Clause: Motivation

- Once a try block is entered, statements in a finally clause are guaranteed to be executed, no matter if an exception is thrown or not.

- Example:

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close();
```

May not get up to here

- `reader.close()` should be executed no matter if exception is thrown or not.
  -> use finally clause for this purpose.

Finally Clause: Example

```
FileReader reader = new FileReader(filename);

try {
  Scanner in = new Scanner(reader);
  readData(in);

} finally {

  reader.close();

}
```

if an exception occurs, finally clause is also executed **before** exception is passed to its handler

Finally Clause

- The finally clause is executed when try block is exited in any of three ways:
  - After last statement of try block.
  - After last statement of catch clause, if this try block caught an exception.
  - When an exception was thrown in try block and not caught.

- Recommendation:
  - keep your code readable
  - don't mix catch and finally clauses in same try block.
  - Use nested try blocks.

Exercise

Which output produces the following program?

```
public static void main(String [] args)
{   try
    {   badMethod();
        System.out.print("A");
    }
    catch (RuntimeException ex)
    {   System.out.print("B");
    }
    catch (Exception ex1)
    {   System.out.print("C");
    }
    finally
    {   System.out.print("D");
    }
    System.out.print("E");
}
public static void badMethod()
{   throw new RuntimeException();
}
```

try-with-resources statement

- Available in Java 7/8
- The try-with-resources statement is a try statement that declares one or more resources.
- A resource is an object that must be closed after the program is finished with it.
- The try-with-resources statement ensures that each resource is closed at the end of the statement.
- Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.
- Replaces the `finally` statement that calls the `close()` method

try-with-resources statement

- Example PrintWriter …

```
try {
  PrintWriter out =
     new PrintWriter(
      new FileWriter(
        "OutFile.txt"));

   for (int i = 0; i < 100; i++) {
      out.println(i);
   }
   out.close();

} catch (IOException e) {
   e.printStackTrace();
}
```

```
…
try (
   PrintWriter out =
     new PrintWriter(
       new FileWriter(
          "OutFile.txt"))) {

   for (int i = 0; i < 100; i++) {
      out.println(i);
   }
  // out.close()

} catch (IOException e) {
   e.printStackTrace();
}
```

Not needed anymore, but still can throw exceptions

try-with-resources statement

Example FileReader …

```
try {
  InputStream in =
    new FileInputStream(
     "OutFile.txt");

  try {
    …
    int next = in.read();
    …
  } finally {
    in.close();
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

```
…
try (
  InputStream in =
    new FileInputStream(
      "OutFile.txt")) {

  // no nested try
  …
  int next = in.read();
  …
  // no finally
  // no close()

} catch (IOException e) {
  e.printStackTrace();
}
```

Outline - Exceptions

- What are exceptions?
- Throwing exceptions
- Exception Handling
- Finally Clause
- Self-made Exception Types

Self-made Exception Types

- You can design your own exception classes as subclasses of
  - Exception (checked) or
  - RuntimeException (unchecked)

Self-made Exception Types

- Example:

```
if (amount > balance) {
 throw new InsufficientFundsException("withdrawal of "
          + amount +" exceeds balance of " + balance);
}
```

- Make this exception an unchecked exception
  - Programmer could have avoided it by calling *getBalance()* first.

- Extend `RuntimeException` or one of its subclasses.
- Supply two constructors
  - Default constructor.
  - A constructor that accepts a message string describing reason for exception.

Self-made Exception Types

- Code for defining this exception

```
public class InsufficientFundsException extends RuntimeException {

    public InsufficientFundsException(){}

    public InsufficientFundsException(String message){
     super(message);
   }
 }
```

Comprehensive Example

- **Non trivial program to show separation between error detection and error handling**
- The program
  - Asks user for name of a file that contains doubles values; reads the values; computes the sum.
  - File format
    - First line of file contains total number of values
    - Remaining lines contain values one per line.
  - Typical input file:

    3

    1.45

    -2.1

    0.05

Comprehensive Example

- Demo
  - DataAnalyzer.java
  - DataSetReader.java
  - BadDataException.java

Summary 1/2

- **Java's exception handling concept**
  - Throw an exception if normal program flow is disrupted.
  - Forward the exception upward the hierarchy in the call stack.
  - Catch the exception at a level where you can handle the situation.
    - Start a dialog with the user, etc.

- Checked vrs. unchecked exceptions
  - **Checked exceptions** are exceptions that a well-written application should anticipate and recover from. Checked exceptions either need to be caught or need to be explicitly listed in method declarations to enable forwarding up the call stack.
  - **Unchecked exceptions** are errors (external) and run time exceptions (internal). Unchecked exceptions need not to be explicitly listed in method declarations.

Summary

- **finally clause**
  - used perform clean up operation before program termination, no matter whether this termination is regular or by exception.
  - In Java 7/8 use try-with-resources statement

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

XML – Part A: Introduction, Specification

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS2018

<?xml?>

- Computer science

Outline – XML

- What is XML?
- Specifying XML Documents

Outline – XML

- What is XML?
- Specifying XML Documents

XML main characteristics

- It is a textual representation of data and/or text containing markup information.

- As opposed to formatting languages,

  this markup names the text elements or data and

  describes their meaning

  rather than their formatting (as for example in HTML).

Hello World

```
<?xml version="1.0"?>
<world>
   <hello>Hello World!!!</hello>
</world>
```

Header

Nested structure: Start and end tags

A first example

An address of a person looks like the following example:

John Smith
Master of Arts
12, 45th Avenue, Apt #15
New York, NY 10003
USA

Human beings easily understand that the first line contains the name, the subsequent one „apparently“ an academic title, the next one a street and an apartment number, and finally a city, followed by its ZIP code, and by a country indication.

However, this ordering isn’t applicable everywhere (cf. Russia = ± inverse order, Europe: ZIP in front of city etc.).

If this information has to be treated automatically, we need some means to recognize formally the parts of this address.

A first example

Using XML this address may be written as follows:

```
<address>
  <firstname>John</firstname> <name>Smith</name>
  <title>Master of Arts</title>
  <street>45th Avenue</street> <number>12</number>
  <apartment-number>15</apartment-number>
  <city>New York</city> <zip>NY 10003</zip>
  <country>USA</country>
</address>
```

What's going on here?

- Contiguous data/values are grouped by <xxx> … </xxx>
- Such structures may be nested
- The names in the <xxx> tags may be selected by the user; the only condition is that the tag's content <xxx> must be the same as in </xxx>.

Basic XML Notions

<xxx> is called a *tag.*
<xxx> is the *start tag*, </xxx> the corresponding *end tag*

A corresponding pair of <xxx> … </xxx> including their enclosed content is an *element.*

If a corresponding pair of <xxx> … </xxx> contains neither text nor other elements, it can be written as <xxx/> (empty element)

The whole ensemble is called a *document*; thus, a document consists of one single element (the *root element*) with its contents

(However, a complete document contains some formal elements in addition.)

XML Naming Conventions

XML elements have to be named according the following rules

- Names start with letters or underscores (_)
- After the first character, numbers, hyphens (-), underscores (_) and periods (.) are allowed
- Names can’t contain spaces
- Names can’t contain the colon (:)
- Names can’t start with the letters xml, in uppercase, lowercase or mixed

XML Naming Conventions

Name	Correct	Not correct	Why?
<first.name>			
<xml-tag>			
<123>			
<a-b>			
<résumé>			
<fun=xml>			
<my tag>			
<CaMeLcAsE>			

XML Attributes

Values may not only be represented as the *content* of a *tag* but also as an *attribute*

```
<book signature="DR20-4537" >
  <author>
      <firstname>Daniel</firstname>
      <name>Defoe</name>
  </author>
   <title>Robinson Crusoe</title>
   <chapter title="Embarking" >
     Here starts the text of the first chapter.
   </chapter>
   <chapter title="Afloat" >
     This is the second chapter.
   </chapter>
   ...
</book>
```

Note:
Attribute
Values must
be quoted !!!

Tag content ↔ Attribute ?
No fixed rule; empirical decision

XML Comments

Between the data elements we may also insert comments:

```
<book signature="DR20-4537" >
  <!-- Data about the book as a whole -->
  <author>
     <firstname>Daniel</firstname>
     <name>Defoe</name>
  <author>
  <title>Robinson Crusoe</title>

  <!-- Text of the book -->
  <chapter title="Embarking" >
    Here starts the text of the first chapter.
  </chapter>
  <chapter title="Afloat" >
    This is the second chapter.
  </chapter>
  ...
</book>
```

The XML Header

At the beginning of each xml document at least the following header line must be added:

```
<?xml version="1.0" encoding="UTF-8"?>
<book signature="DR20-4537" >
   <!-- Data about the book as a whole -->
   <author>
      <firstname>Daniel</firstname>
       <name>Defoe</name>
   </author>
    <title>Robinson Crusoe</title>

   <!-- Text of the book -->
   <chapter title="Embarking" >
    Here starts the text of the first chapter.
   </chapter>
   <chapter title="Afloat" >
    This is the second chapter.
   </chapter>
    ...
</book>
```

The XML Header

```
<?xml version="1.0" encoding="UTF-8"
                                  standalone="no"?>
```

There are versions 1.0 and 1.1;
however, v 1.1 is not well accepted,
And v 1.0 is still preferred.

Rarely used;
Values are yes or no (default).
Yes means there is no external DTD.

There are several encoding standards;

- UTF-8 and UTF-16: Standardized by ISO/IETF, used worldwide, represents full Unicode, using variable-length multibyte coding
- ISO-8859-1: used mostly in US, ≈ extended ASCII
- Shift-JIS: used mostly in Japan, for Japanese alphabet.

Be aware of your editors or processing programs since they might tacitly assume a (different!!) standard!

Exercise BankAccount

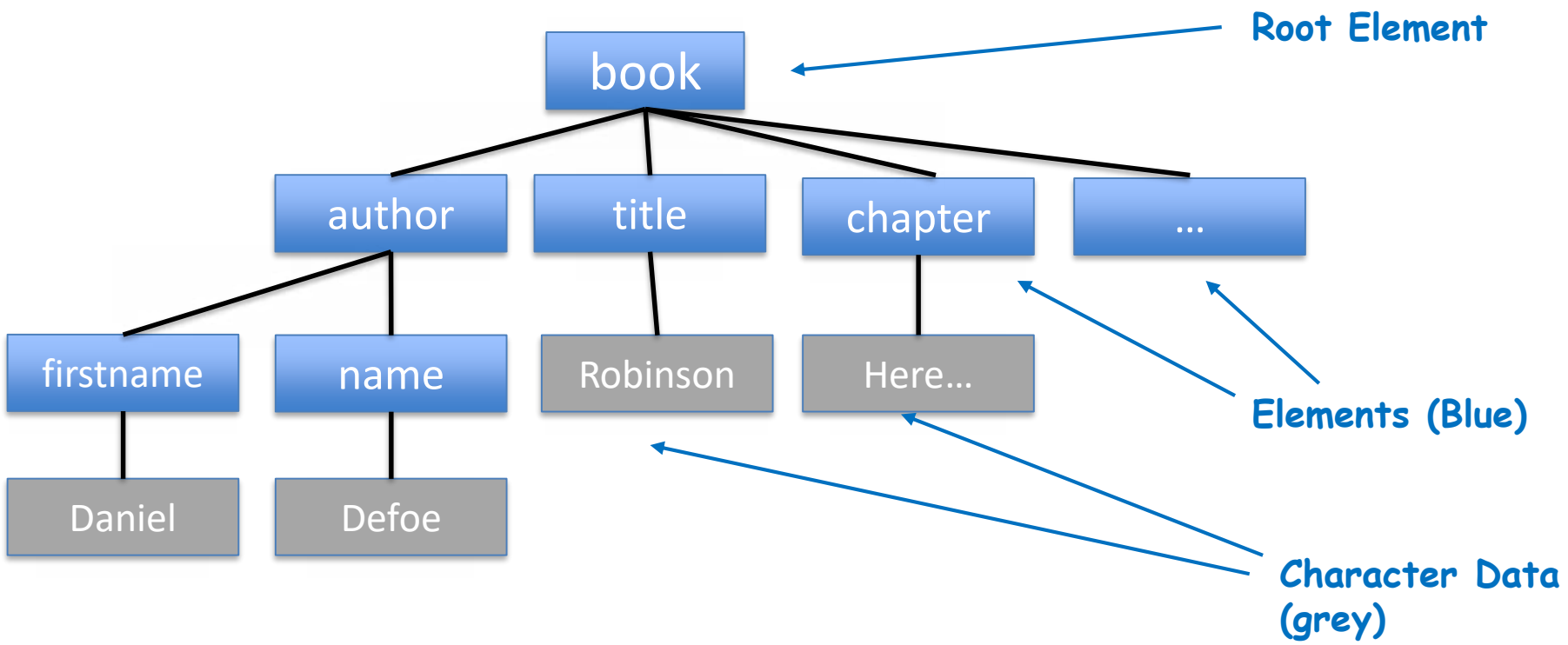
Write down the data characterizing a bank account as an XML document.

The following data should be contained:

- account number
- name and address of the owner
- account status
- type of account (current account, savings account etc.).

If you see further data to be enclosed … go ahead!

XML documents have a tree structure



An XML document is an **ordered, labeled tree**

XML documents have a tree structure

An XML document is an **ordered, labeled tree**, consisting of

- **elements** nodes, each labeled with
  - a name (often called the element *type*),
  - (optionally) a set of **attributes**, each consisting of a name and a value,
  - and these nodes can have child nodes

- **character data** leaf nodes contain the actual data (text strings)
  - normally, character data nodes are **non-empty** and **non-adjacent to other character data nodes**

Representing Relations between Data

Representation of data

- text values between element tags, or
- values of attributes

Representing relations between data =

- **Sequence**

sequence = equivalent + ordered

```
<author>Anthony First</author>
<author>John Second</author>
```

- **Nesting**

nesting = „is part of"

```
<author>
  <email>Maggie.Herrick@bbs.mhv.net</email>
</author>
```

XML Entities

What if you want to write a value containing a `<` sign, e.g.

```
<condition> e < 2 </condition> ?
```

XML defines some standard ***entities*** to avoid this problem:

`<`	`<`,	a „less than“ sign
`>`	`>`,	a „greater than“ sign
`"`	`"`,	a „double quote“ sign
`'`	`'`,	a „single quote“ or „apostroph“ sign
`&`	`&`,	an „ampersand“ sign

```
<condition> e &lt; 2 </condition>
```

XML processors (→ XSLT) will automatically convert these entities to the actual characters for output but NOT treat them according to their normal XML meaning
Users may define their own entities (→ DTD/Schema)

Namespaces

What‘s the Problem?

Since the tag names and document structures may be defined specifically for each application, the **same tag names** might get used to denote **different things**. Joining such documents will lead to clashes:

Geometry:	<plane> … </plane>
Air traffic:	<plane> … </plane>
Geography:	<plane> … </plane>

Namespaces

The solution

In order to avoid such name clashes, we define name spaces and we prefix each tag name with the namespace indicator:

Geometry: <geom:plane> … </geom:plane>

Air traffic: <fly:plane> … </fly:plane>

Geography: <geog:plane> … </geog:plane>

HOWEVER:
These namespace indicators (geom, fly …) really must be different! How do we meet this requirement?

Namespaces

How can we assure disjunct namespaces ?

Using „geog“ is just a shorthand form for a longer name, and it is this longer name which really must be different from any other. The idea is to use URLs from the Internet which indeed are guaranteed to be unique:

```
<?xml version="1.0" encoding="UTF-8"?>
<flight-geometry
  xmlns:geom="http://www.mycompany.com/homepage/subpage"
  xmlns:fly="http://www.anyairline.com/homepage/subpage">
...
```

Shorthand name of the namespace

Any/my existing URL as a guarantee for uniqueness...

... possible extended by some (fictitious) page/subpage name

Namespaces

An example of the usage of namespaces:

```
<?xml version="1.0" encoding="UTF-8"?>
<flight-geometry
  xmlns:geog="http://www.belpmoos.com/homepage/subpage"
  xmlns:fly="http://www.anyairline.com/homepage/subpage">

  <rule>
    The Berne airport is located in the <geog:plane> Belpmoos
    </geog:plane>. <fly:plane> Jets </fly:plane> are always
    landing from North while <fly:plane> small planes
    </fly:plane> may land from both sides.
  </rule>
  …
</flight-geometry>
```

Default Namespaces

When many elements use the same namespace

→ Define a default namespace: attribute **`xmlns`**.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<flight-geometry
  xmlns="http://www.belpmoos.com/homepage/subpage"
  xmlns:fly="http://www.anyairline.com/homepage/subpage">

  <rule>
    The Berne airport is located in the <plane> Belpmoos
    <plane>. <fly:plane> Jets </fly:plane> are always
    landing from North while <fly:plane> small planes
    </fly:plane> may land from both sides.
  </rule>
  …
```

Links to more information

- Internet
  - **www.w3.org/TR/REC-xml.html**
    the XML 1.0 specification
  - **www.w3.org/TR/xml11**
    XML 1.1 (Candidate Recommendation), minor changes to reflect Unicode revisi
  - **www.w3.org/XML**
    W3C's XML homepage
  - **www.xml.com**
    XML information by O'Reilly: articles, software, tutorials
  - **www.oasis-open.org/cover**
    The XML Cover Pages: comprehensive online reference
  - **www.w3schools.com/xml**
    XML School: an XML tutorial
- Books
  - David Hunter et al.: **Beginning XML (Programmer to Programmer)** Wrox.
    (An excellent introductory book.)
  - Elliotte Rusty Harold, W. Scott Means: **XML in a Nutshell, Third Edition**. O'Re
  - Daniel Koch: **XML für Webentwickler**. Ein praktischer Einstieg. Hanser 2010

Exercise Literature

Correct the given XML document.

```
<?xml version="1.0">
<literature>
<-- First reference -->
 <reference number:"1">
   <author>
    <name>Jefferson<firstname>Ron</name></firstname>
   </author>
   <title>XML for Beginners<title/>
   <editor>McGraw Hill</editor>
   <isbn>12-3344-567-00</ibsn>
  </Reference>
</literature>
```

Outline – XML

- What is XML?
- Specifying XML Documents

Specifying XML Documents

In order to be treated correctly, XML documents must be

- **well-formed**, and
- **valid**.

well-formed : Each opening tag has its corresponding end tag,

```
<street>Main Street<number>24</street>
```

… and tags are strictly nested

```
<street>Main Street<number>24</street></number>
```

valid : The tag names, their nesting, the value types etc. are specified by a specification

Specifying XML Documents

XML Documents may be specified using 2 different techniques:

- DTD (Document Type Definition)

  Outdated!

  A Document Type Definition defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements.
  A DTD can be declared inline in your XML document, or as an external reference.

- XML Schema Definition

  The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

  An XML Schema:
  - defines elements that can appear in a document
  - defines attributes that can appear in a document
  - defines which elements are child elements
  - defines the order of child elements
  - defines the number of child elements
  - defines whether an element is empty or can include text
  - defines data types for elements and attributes
  - defines default and fixed values for elements and attributes

Specifying an XML Document using an XML Schema

A Schema is always an external file which must be referenced by the **`schemaLocation`** attribute:

```
<address
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="“http://example.org/myself/address.xsd">
    <firstname>John</firstname>
    <name>Smith</name>
    <title>Master of Arts</title>
    <street>45th Avenue</street>
    <number>12</number>
    <apt-number>15</apt-number>
    <city>New York</city>
    <zip>10003</zip>
    <country>USA</country>
</address>
```

XML Schema

Thus, let‘s start …

```
<?xml version="1.0" encoding...
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 ...
</xsd:schema>
```

This URL is mandatory.
Though there has been a new version in 2004, the XML world insists on the 2001 version!

XML Schema: Elements

a) Simple Elements

any standard XML type ☞☞☞

```
<xsd:element name="firstname" type="xsd:string"/>
```

or

```
<xsd:element name="firstname" type="xsd:string" default="Joe"/>
<xsd:element name="firstname" type="xsd:string" fixed="Bob"/>
```

default or fixed values

XML Schema: Standard Types

In XML, standard types are defined for

- strings/tokens

  `xsd:string, xsd:token`

- Date, Time, DateTime and Duration types

  `xsd:date, xsd:time, xsd:duration, xsd:gDay, gMonthDay...`

- Numbers (Decimal ≈ double, integer numbers from byte … long)

  `xsd:decimal, xsd:integer, xsd:unsignedLong, xsd:byte...`

- boolean

  `xsd:boolean`

- binary data

  `xsd:hexBinary, xsd:base64Binary`

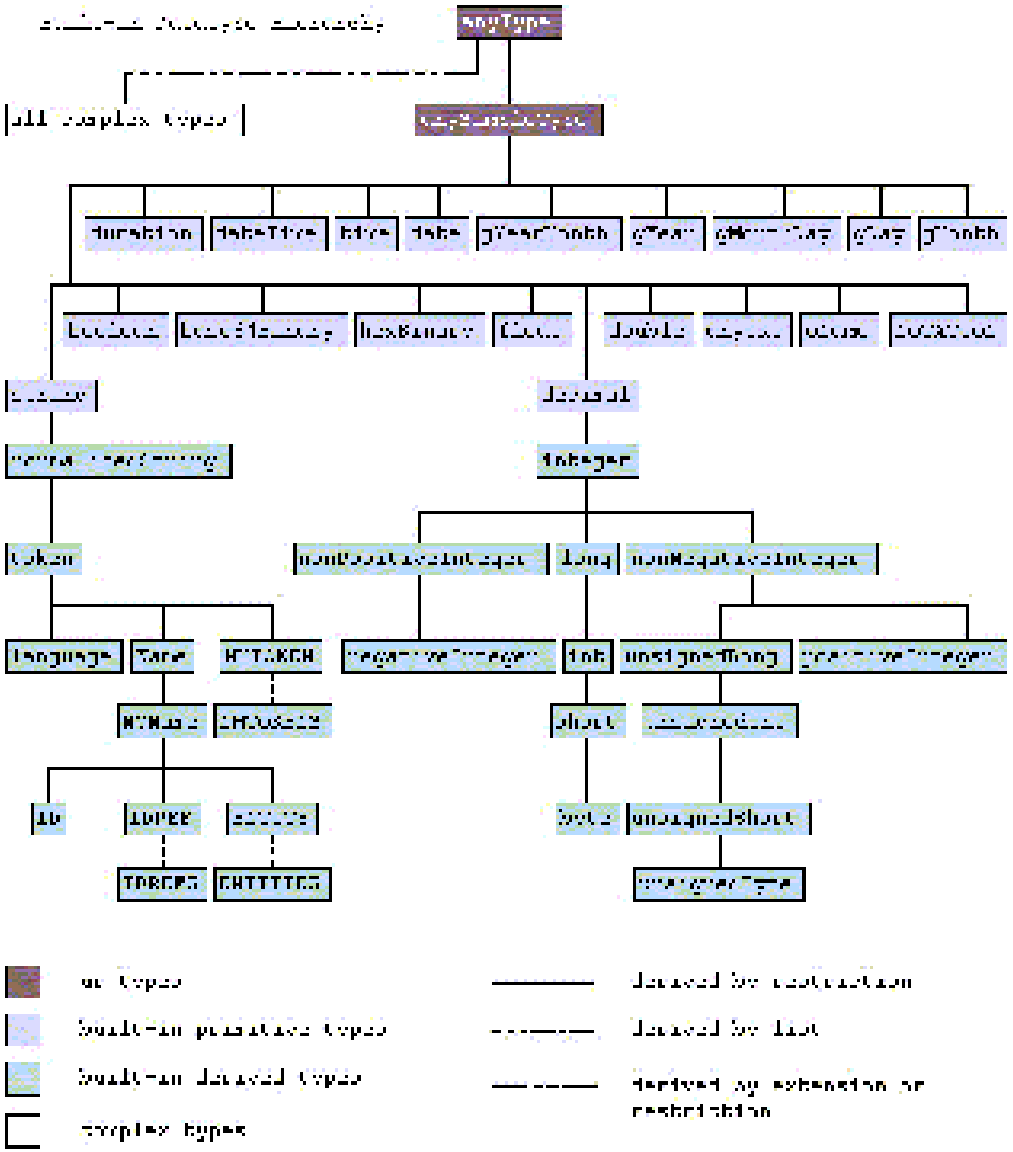
- URI

  `xsd:anyURI`

- ...

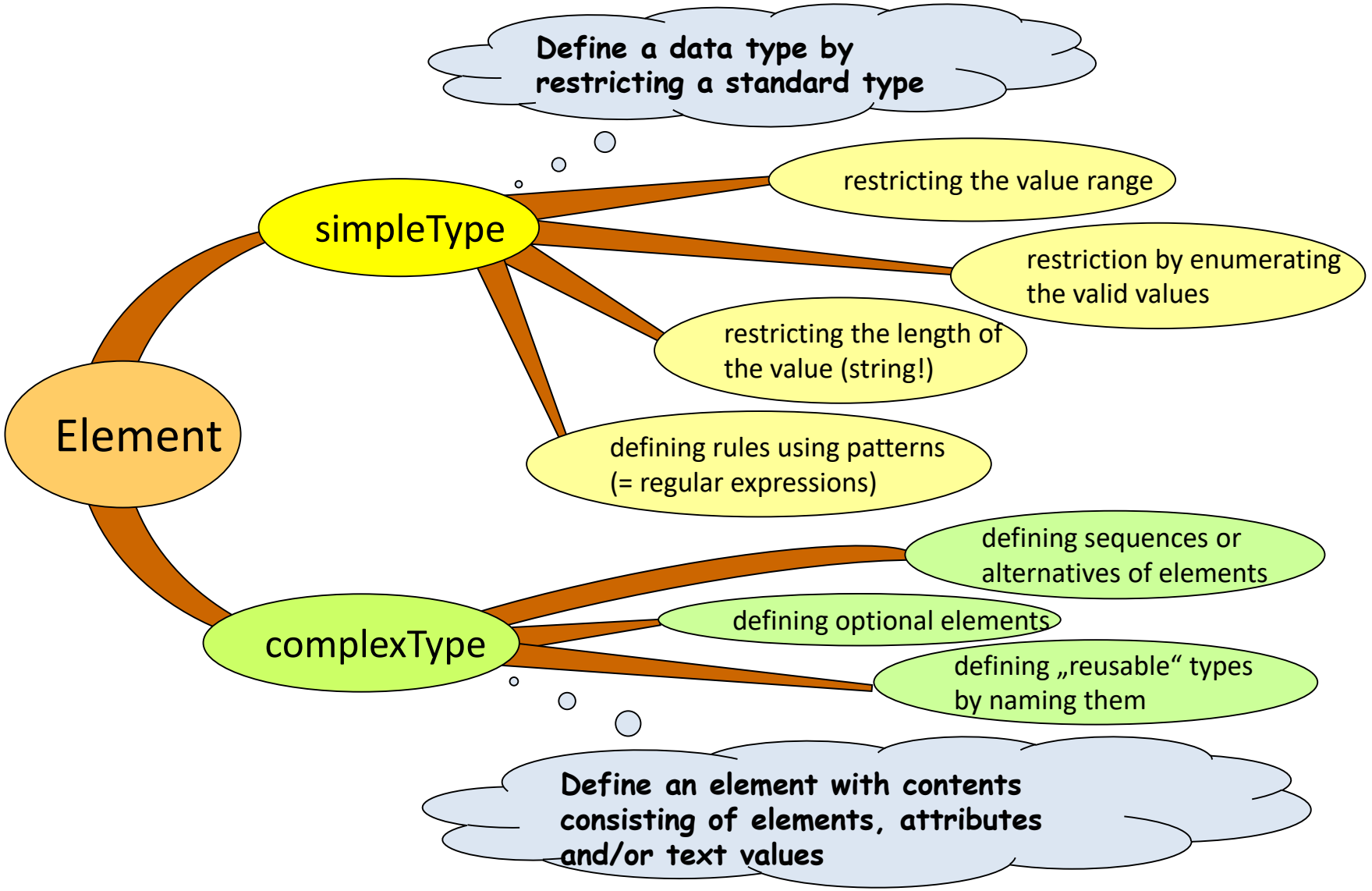
see separate sheet

XML Schema: Standard Types



W3C
XML Schema Part 2
Datatypes

XML Schema: Typed Elements



XML Schema: simpleType Elements – Value Range

simpleType

- Restrictions on Value Ranges

```
<xsd:element name="age">
  <xsd:simpleType>
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="0"/>
      <xsd:maxInclusive value="150"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Indicates which basic type is used and restricted

Defining a restricted interval, for example

- Also **`min/maxExclusive`** possible
- Example xml:

```
<age>75</age>
```

<age> is restricted to values between 0 und 150

XML Schema: simpleType Elements – Set of Values

simpleType

- Restrictions on a Set of Values

```
<xsd:element name="car">
 <xsd:simpleType>
    <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Audi"/>
       <xsd:enumeration value="VW"/>
       <xsd:enumeration value="BMW"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Defining a list of admitted values

- Example xml:

```
<car>Audi</car>
```

<car> is restricted to 3 values

XML Schema: simpleType Elements - Patterns

simpleType

- Restrictions on a Series of Values (Patterns)

```
<xsd:element name="letter">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
       <xsd:pattern value="[a-z]*"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Indicates which basic type is used and restricted

Only one lower case letter allowed

- Example xml:

```
<letter>abc</letter>
```

<letter> is restricted to lower cases

XML Schema: simpleType Elements - Trim

simpleType

- Remove Extra White Space Characters

```
<xsd:element name="singleSpace">
  <xsd:simpleType>
     <xsd:restriction base="xsd:string">
        <xsd:whiteSpace value="collapse"/>
     </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Each contiguous group of white space characters will be collapsed into one single space

- Example xml:

```
<singleSpace>
this  is  a   sentence
</singleSpace>
```

White spaces collapsed when processed

XML Schema: complexType Elements 1/4

complexType

- Each element containing either other elements or text values, or having attributes is a **complexType** element.
- Contents may be sequences, alternatives and optional elements, all characterized by a cardinality.

```
<xsd:element name="employee">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="firstname" type="xsd:string"/>
      <xsd:element name="lastname"  type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

XML Schema: complexType Elements 2/4

complexType

`xsd:sequence`: You may specify the **number of occurrences** of enclosed elements;

```
<xsd:element name="employee">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="prenom" type="xsd:string"
        minOccurs="1" maxOccurs="5"/>
      <xsd:element name="nom" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

minOccurs = maxOccurs = 1
if not specified

XML Schema: complexType Elements 3/4

complexType

`xsd:choice:` You may choose among **alternative elements**; :

```
<xsd:element name="note">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="to" type="xsd:string"/>
      <xsd:element name="from" type="xsd:string"/>
      <xsd:element name="header" type="xsd:string"/>
      <xsd:element name="message" type="xsd:string"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Exactly one of the elements listed within "choice" may appear as a child element of "note"

XML Schema: complexType Elements 4/4

complexType

xsd:all: You may specify that all elements must appear exactly once but in an arbitrary order:

```
<xsd:element name="note">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="to" type="xsd:string"/>
      <xsd:element name="from" type="xsd:string"/>
      <xsd:element name="header" type="xsd:string"/>
      <xsd:element name="message" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>
```

Each of the elements listed within "all" may appear just once but in any order

XML Schema: Defining Attributes 1/3

attribute

Attributes are defined similarly to simple elements:

```
<xsd:attribute name="xxxx" type="yyyy"/>
```

Example: In the element

```
<lastname lang="DE"> ... </lastname>
```

the **lang** attribute is defined as:

```
<xsd:element name="lastname">
  <xsd:complexType>
    ...
    <xsd:attribute name="lang" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
```

As soon as an element has an attribute, it is considered to have a `complexType`

Attribute definitions AFTER element definitions

XML Schema: Defining Attributes 2/3

attribute

As with simple elements, the values of attributes may be defined as default or fixed values:

```
<xsd:attribute name="xxxx" type="yyyy" default="zzzz"/>
<xsd:attribute name="xxxx" type="yyyy" fixed="ffff"/>
```

`Fixed` means that the user is not allowed to specify a value himself.

Example: In the element

```
<productname lang="FR"> ... </productname>
```

we could preset the **lang** attribute as:

```
<xsd:element name=«productname">
  <xsd:complexType>
    ...
    <xsd:attribute name="lang" type="xsd:string" default="EN"/>
  </xsd:complexType>
</xsd:element>
```

If the document explicitly specifies a value for `lang`, this value is used; otherwise, `EN` will be used.

XML Schema: Defining Attributes 3/3

attribute

- By default, attributes are assumed to be optional.
- If you want an attribute to be mandatory, add the attribute „use“:

```
<xsd:attribute name="xxxx" type="yyyy" use="required"/>
```

The attribute `use` may have the value `required` or `optional` which is the default value.

```
<productname lang="FR"> ... </productname>
```

XML Schema: Combining All Kinds of Nodes 1/3

Text values can be used in several ways:

1. An element contains only a single text value:

   → The element is declared as a *simpleElement*:

```
<xsd:element name="firstname" type="xsd:string"/>
```

Example:

```
<firstname>John</firstname>
```

XML Schema: Combining All Kinds of Nodes 2/3

- 2. An element contains text but also an attribute:

Specifying an attribute always requires a complexType.

```
<xsd:element name="shoesize">
  <xsd:complexType mixed="true">
    <xsd:attribute name="country" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
```

Text + anything else requires attribute mixed="true"

This defines the attribute.

Example:

```
<shoesize country="france">35</shoesize>
```

XML Schema: Combining All Kinds of Nodes 3/3

3. An element contains intermixed text values and elements (and attributes):

```
<xsd:element name="letter">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="orderid" type="xsd:integer"/>
      <xsd:element name="shipdate" type="xsd:date"/>
    </xsd:sequence>
    <xsd:attribute name="format" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
```

allows for mixing text and elements

elements contained in `letter`

any attributes go here (after the sequence!)

Example:

```
<letter format="A4">
Dear Mr. <name>John Smith</name>,
Your order <orderid>1032</orderid>
will be shipped on <shipdate>2007-01-05</shipdate>
</letter>
```

Exercise Hotel

Write an XML Schema for the following XML document.

Hint: Use the following tool to generate the xsd skeleton:
https://www.freeformatter.com/xsd-generator.html . Use the «Russian Doll» format.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
  <!-- Restrict the stars to values between 1 and 7 -->
  <hotel stars="3">
      <!-- each element should occur only once -->
      <!-- only positive numbers are allowed -->
      <single-room> 12 </single-room>
      <double-room> 26 </double-room>
      <suite> 2 </suite>
      <!-- optional empty element -->
      <lounge/>
</hotel>
```

XML Schema: An Overall Example 1/3

We want to define a schema for the following document:

```
<book isbn="123-456-789" year-of-issue="1699">
  <title>King Lear</title>
  <author>William Shakespeare</author>
  <character>
    <name>King Lear</name>
    <age>75</age>
  </character>
  <character>
    <name>XYZ</name>
    <age>25</age>
  </character>
</book>
```

XML Schema: An Overall Example 2/3

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="title"/>
        <xs:element type="xs:string" name="author"/>
        <xs:element name="character" maxOccurs="unbounded" minOccurs="“1">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="name"/>
              <xs:element type="xs:byte" name="age"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="isbn"/>
      <xs:attribute type="xs:gYear" name="year-of-issue"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema: An Overall Example 3/3

A „Russian Doll“ example:

- nested structures
- structure similar to the specified XML document
- relations between schema parts described by nesting:
  - no possibility to „re-use“ parts of the schema
  - not easily readable
  - not easily maintainable

XML Schema: Naming and Referencing Types (1/2)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!--  Root element -->
  <xs:element name="book" type="bookType"/>

  <!--  Defining simple types -->
  <xs:simpleType name="nameType">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="32"/>
    </xs:restriction>
  </xs:simpleType>
   …
```

The entire document is defined by one single (root) element!

A type definition for a simpleType

XML Schema: Naming and Referencing Types (2/2)

```
…
<!--  Defining complex types -->
<xs:complexType name="characterType">
  <xs:sequence>
    <xs:element type="nameType" name="name"/>
    <xs:element type="xs:byte" name="age"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="bookType">
  <xs:sequence>
    <xs:element type="nameType" name="title"/>
    <xs:element type="nameType" name="author"/>
    <xs:element type="characterType" name="character" maxOccurs="unbounded" minOccurs="1"/>
  </xs:sequence>
  <xs:attribute type="xs:string" name="isbn"/>
  <xs:attribute type="xs:gYear" name="year-of-issue"/>
</xs:complexType>
</xs:schema>
```

More type definitions for complexTypes

Exercise Hotel 2

Once more: Regenerate the Scheme in the «Venetian Blind» format.

Write a specific type definition for the stars attribute allowing only 1 .. 5 stars.

```
<?xml version="1.0" encoding="UTF-8" ?>
<hotel stars="***">
  <single-room> 12 </single-room>
  <double-room> 26 </double-room>
  <suite> 2 </suite>
  <!-- optional element -->
  <lounge/>
</hotel>
```

XML Schema: Defining a Namespace for a Schema

Namespaces can be used in two ways:

- Defining our own namespace for the schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://example.org/myself/myschema"
            xmlns="http://example.org/myself/myschema"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
...
</xsd:schema>
```

The standard XMLSchema namespace to use for the schema definition itself.

In order to use this namespace here, we declare it as the default namespace within this schema.
This means that we don't have to write a prefix for our own (element/attribute…) definitions

Everybody writing documents according to this schema must use this namespace.
If no **targetNamespace** is specified, **noNamespace** will be used.

All newly defined elements belong to the namespace.

XML Schema: Defining a Namespace for a Schema

- Using one or more namespaces defined elsewhere

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:other="http://somebody.else.org/someschema">
  ...
</xsd:schema>
```

Again:
The standard namespace to use
for the schema definition itself

Using an already existing namespace

Use of Namespace

- Namespaces are a mechanism for breaking up your schemas.
- Up until now, we had only a single schema file containing all element definitions,
- but the XSD standard allows you to structure your XSD schemas by breaking them into multiple files.

Example: In this example, the schema is broken out into 4 files.

- **CommonTypes** - this could contain all your basic types, AddressType, PriceType, PaymentMethodType etc.
- **CustomerTypes** - this could contain all your definitions for your customers.
- **OrderType** - this could contain the definitions for orders.
- **Main** - this would pull all the sub schemas together into a single schema, and define your main element/s.

Example: CommonTypes.xsd 1/2

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    targetNamespace="http://NamespaceTest.com/CommonTypes"
    xmlns="http://NamespaceTest.com/CommonTypes">

    <xsd:complexType name="AddressType">
        <xsd:sequence>
            <xsd:element name="Line1" type="xsd:string" />
            <xsd:element name="Line2" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
```

Unique Identifier as targetNamespace

Reusable item (type)

Example: CommonTypes.xsd 2/2

```
    <xsd:simpleType name="PriceType">
        <xsd:restriction base="xsd:decimal">
            <xsd:fractionDigits value="2" />
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="PaymentMethodType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="VISA" />
            <xsd:enumeration value="MasterCard" />
            <xsd:enumeration value="Cash" />
            <xsd:enumeration value="Amex" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Reusable items (types)

Example: CustomTypes.xsd

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://NamespaceTest.com/CustomerTypes"
    xmlns:cmn="http://NamespaceTest.com/CommonTypes"
    elementFormDefault="qualified">

    <xsd:import schemaLocation="CommonTypes.xsd" namespace="http://NamespaceTest.com/CommonTypes" />

    <xsd:complexType name="CustomerType">
        <xsd:sequence>
            <xsd:element name="Name" type="xsd:string" />
            <xsd:element name="DeliveryAddress" type="cmn:AddressType" />
            <xsd:element name="BillingAddress" type="cmn:AddressType" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Unique Identifier as targetNamespace

Import of another schema

Use of type from another schema

Example: OrderType.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://NamespaceTest.com/OrderType"
    xmlns:cmn="http://NamespaceTest.com/CommonTypes"
    elementFormDefault="qualified">

<xsd:import namespace=http://NamespaceTest.com/CommonTypes  schemaLocation="CommonTypes.xsd" />

<xsd:complexType name="OrderType">
    <xsd:sequence>
        <xsd:element maxOccurs="unbounded" name="Item">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="ProductName" type="xsd:string" />
                    <xsd:element name="Quantity" type="xsd:int" />
                    <xsd:element name="UnitPrice" type="cmn:PriceType" />
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Unique Identifier as targetNamespace

Import of another schema

Use of type from another schema

63

Example: Main.xsd 1/2

Unique Identifier as targetNamespace

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://NamespaceTest.com/Purchase"
    xmlns:ord="http://NamespaceTest.com/OrderType"
    xmlns:cmn="http://NamespaceTest.com/CommonTypes"
    xmlns:cust="http://NamespaceTest.com/CustomerTypes"
    elementFormDefault="qualified">
```

Definition of alias

```
<xsd:import schemaLocation="CommonTypes.xsd" namespace="http://NamespaceTest.com/CommonTypes" />
<xsd:import schemaLocation="CustomerTypes.xsd" namespace="http://NamespaceTest.com/CustomerTypes" />
<xsd:import schemaLocation="OrderType.xsd" namespace="http://NamespaceTest.com/OrderType" />
```

Import of 3 used schemata

Example: Main.xsd 2/2

```
  <xsd:element name="Purchase">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDetail" type="ord:OrderType" />
        <xsd:element name="PaymentMethod" type="cmn:PaymentMethodType" />
        <xsd:element name="CustomerDetails" type="cust:CustomerType" />
     </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

</xsd:schema>
```

Use of types from other schemata with different alias

65

Example: Purchase.xml 1/2

```
<?xml version="1.0"?>
<p:Purchase
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://NamespaceTest.com/Purchase Main.xsd"
    xmlns:p="http://NamespaceTest.com/Purchase"
    xmlns:o="http://NamespaceTest.com/OrderType"
    xmlns:c="http://NamespaceTest.com/CustomerTypes"
    xmlns:cmn="http://NamespaceTest.com/CommonTypes">

    <p:OrderDetail>
        <o:Item>
            <o:ProductName>Widget</o:ProductName>
            <o:Quantity>1</o:Quantity>
            <o:UnitPrice>3.42</o:UnitPrice>
        </o:Item>
    </p:OrderDetail>

    <p:PaymentMethod>VISA</p:PaymentMethod>
```

Root element

Schema location

Definition of alias

alias = "p" because defined in Main.xsd

alias = "o" because defined in OrderType.xsd

Example: Purchase.xml 2/2

```
    <p:CustomerDetails>
        <c:Name>James</c:Name>
        <c:DeliveryAddress>
            <cmn:Line1>15 Some Road</cmn:Line1>
            <cmn:Line2>SomeTown</cmn:Line2>
        </c:DeliveryAddress>
        <c:BillingAddress>
            <cmn:Line1>15 Some Road</cmn:Line1>
            <cmn:Line2>SomeTown</cmn:Line2>
        </c:BillingAddress>
    </p:CustomerDetails>
</p:Purchase>
```

alias = "p" because defined in Main.xsd

alias = "c" because defined in CustomerType.xsd

Use of Namespace – general rules

- The alias must be the same as the target namespace in which the **element** is defined.
- It is important to note that this is where the element is defined - not where the *complexType* is defined.
- Example:
  - So the element <OrderDetail> is actually defined in main.xsd so it is part of the namespace "http://NamespaceTest.com/Purchase", even though it uses the complexType "OrderType" which is defined in the OrderTypes.xsd.
  - The contents of <OrderDetail> are defined within the complexType "OrderType", which is in the target namespace "http://NamespaceTest.com/OrderTypes", so the child element <Item> needs qualifying within the namespace "http://NamespaceTest.com/OrderTypes".

Specifying XML Documents

References:

XML Schema Definition

- XML Schema Part 0: Primer Second Edition
  `http://www.w3.org/TR/xmlschema-0/`
- XML Schema Teil 0: Einführung.
  `http://www.edition-w3c.de/TR/2001/REC-xmlschema-0-20010502/`

DTD (Document Type Definition)

- Extensible Markup Language (XML) 1.0 (Second Edition)
  `http://www.w3.org/TR/2000/REC-xml-20001006`
- Dokumenttyp-Definitionen (DTDs)
  `http://de.selfhtml.org/xml/dtd/index.htm`

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

JSON – Basics

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS2018

- Computer science

Outline – JSON

- JSON Basics
- Processing JSON

Outline – JSON

- JSON Basics
- Processing JSON

What is JSON?

- JSON = JavaScript Object Notation
- JSON is a simple, text-based way to store and transmit structured data.
- can store anything from a single number through to strings, arrays, and objects
- can nest arrays and objects, to create complex data structures.

Advantage

JSON has a lot of advantages:

- It's **compact**
- It's **easy** for both computers and people to read and write
  - uses a simple syntax
- It **maps** very easily onto the data structures used by most programming languages (numbers, strings, booleans, nulls, arrays and associative arrays)
- Nearly all programming languages contain functions or libraries that can read and write JSON structures

What is JSON used for?

- used in web applications to send data from the server to the browser.
  - Typically you transfer JSON data using **Ajax** (short for "Asynchronous JavaScript + XML"), which lets your web application exchange data and messages between the browser and the server without having to reload the page.

How to write JSON strings (1/2)

- A **JSON string** contains either an array of values, or an object (an associative array of properties).
  - An **array** is surrounded by square brackets (**[ ]**) and contains a comma-separated list of values.
  - An **object** is surrounded by curly brackets (**{ }**), and contains a comma-separated list of properties (name/value pairs).

How to write JSON strings (2/2)

- A **name/value** pair consists of a field name (in double quotes), followed by a colon (**:**), followed by the field value.
- A **value** in an array or object can be:
  - A **number** (integer or floating point)
  - A **string** (in double quotes)
  - A **boolean** (true or false)
  - Another **array** (surrounded by [ ])
  - Another **object** (surrounded by { })
  - The value **null**

- **Hint**: For more detail, see on **http://json.org/**

JSON Example – Order

```
{
    "orderID": 12345,
    "shopperName": "John Smith",
    "shopperEmail": "johnsmith@example.com",
    "contents": [
        {   "productID": 34,
            "productName": "SuperWidget",
            "quantity": 1
        },
        {   "productID": 56,
            "productName": "WonderWidget",
           "quantity": 3
        }
    ],
    "orderCompleted": true
}
```

A property with name "orderID" and its integer value 12345

A property with name "contents", whose value is an array

Inside the "contents" array, are 2 objects representing order lines. Each object contains 3 properties: productID, productName, and quantity.

Outline – JSON

- JSON Basics
- Processing JSON

Processing JSON with Java

- There a lot of way to process JSON
- See: https://www.json.org/
- No standard until now

- **Google GSON**
  - One of the easiest libraries
  - https://github.com/google/gson
  - Download the jar file on http://repo1.maven.org/maven2/com/google/code/gson/gson/2.8.2/

JSON To Java

```
try (Reader reader =
        new InputStreamReader(
            new InputStreamReader(new FileInputStream("order.json")), "UTF-8"){

      Gson gson = new GsonBuilder().create();
      Order o = gson.fromJson(reader, Order.class);
      System.out.println(o);

} catch (UnsupportedEncodingException e) {
      System.out.println("Something is incorrect with the encoding.");
} catch (IOException e) {
      System.out.println("Something is incorrect with the file.");
      e.printStackTrace();
}
```

Input the file name and encoding

Create the GsonBuilder

Read the JSON File and construct a corresponding JAVA object of the given class

Handle the exceptions

Variants to construct the InputStreamReader

- From a file:

```
new InputStreamReader(new FileInputStream("order.json"), "UTF-8");
```

- From a resource (URL or file):

```
new InputStreamReader(
    JsonToJava.class.getResourceAsStream("order.json"), "UTF-8")
```

- And there are more …

Creating Java objects using Reflection

- **Java Reflection** makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time.
- It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.
- To learn more: http://tutorials.jenkov.com/java-reflection/index.html

- Gson uses reflection to create java instances

Reflection example

Names of fields and JSON properties must match

Java class

```
public class Order {
  private int orderID;
  private String shopperName;
  private String shopperEmail;
  private Boolean orderCompleted;
  private ArrayList<Product> contents;
}

public class Product {
  private Integer productID;
  private String productName;
  private Integer quantity;
}
```

JSON file

```
{"orderID": 12345,
"shopperName": "John Smith",
"shopperEmail": "johnsmith@example.com",
"contents": [
    {  "productID": 34,
       "productName": "SuperWidget",
       "quantity": 1
    },
    {  "productID": 56,
       "productName": "WonderWidget",
       "quantity": 3
    }
],
"orderCompleted": true
}
```

Write JSON

- 2 Steps:
  1. Create the GsonBuilder()
  2. Create JSON Stream and output it
     - To `System.out`

```
gson.toJson(new Product(123, "Product1", 7), System.out);
```

     - Using a `FileWriter`

```
Writer writer = new FileWriter("Output.json");
gson.toJson(new Product(123, "Product1", 7), writer);
```

     - Using a `OutputStreamWriter`

```
Writer writer = new OutputStreamWriter(
    new FileOutputStream("Output.json"), "UTF-8")
gson.toJson(new Product(123, "Product1", 7), writer);
```

Outlook

- JSON schemes and validation are currently under developmenet
  - See: http://json-schema.org/

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

XML – Part B: Processing and Storing

JAXB

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS2018

<?xml?>

- Computer science

Outline – XML B - JAXB

- What is JAXB?
- Marshal and Unmarshal
- Annotating classes
- Generating classes from XSD

Outline – XML B - JAXB

- What is JAXB?
- Marshal and Unmarshal
- Annotating classes
- Generating classes from XSD

JAXB (Java Architecture for XML Binding)

JAXB is a Java standard that defines how Java objects are converted to/from XML (specified using a standard set of mappings).

- JAXB defines a API for **reading** and **writing** Java objects to/from XML documents
- More details: https://github.com/javaee/jaxb-v2

Outline – XML B - JAXB

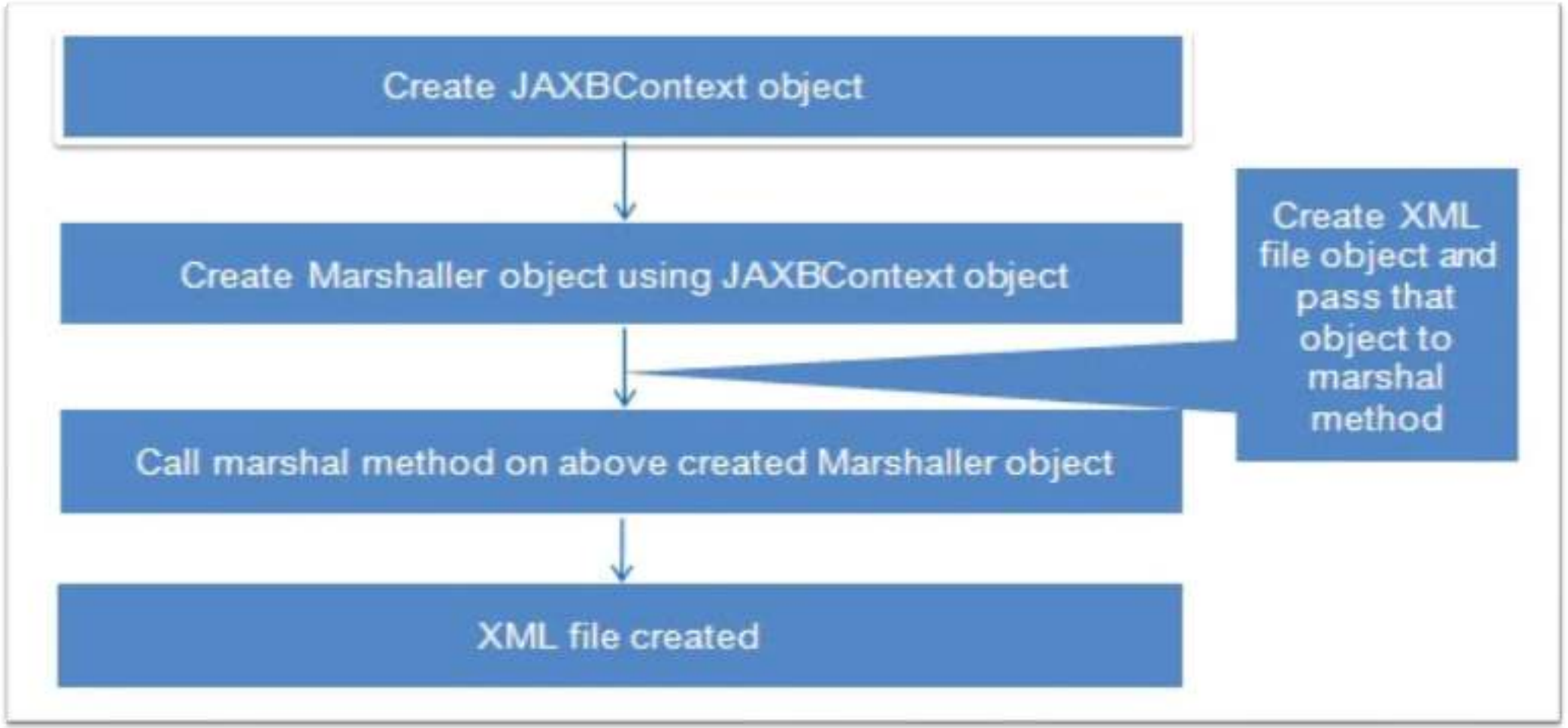
- What is JAXB?
- Marshal and Unmarshal
- Annotating classes
- Generating classes from XSD

Using JAXB

- **With JAXB you can:**
  - **Save** Java Objects as XML file (**marshal**)
  - **Load** an XML file into Java (**unmarshal**)

- Prerequisite: **Annotated java classes !!!**

Using JAXB - Marshalling



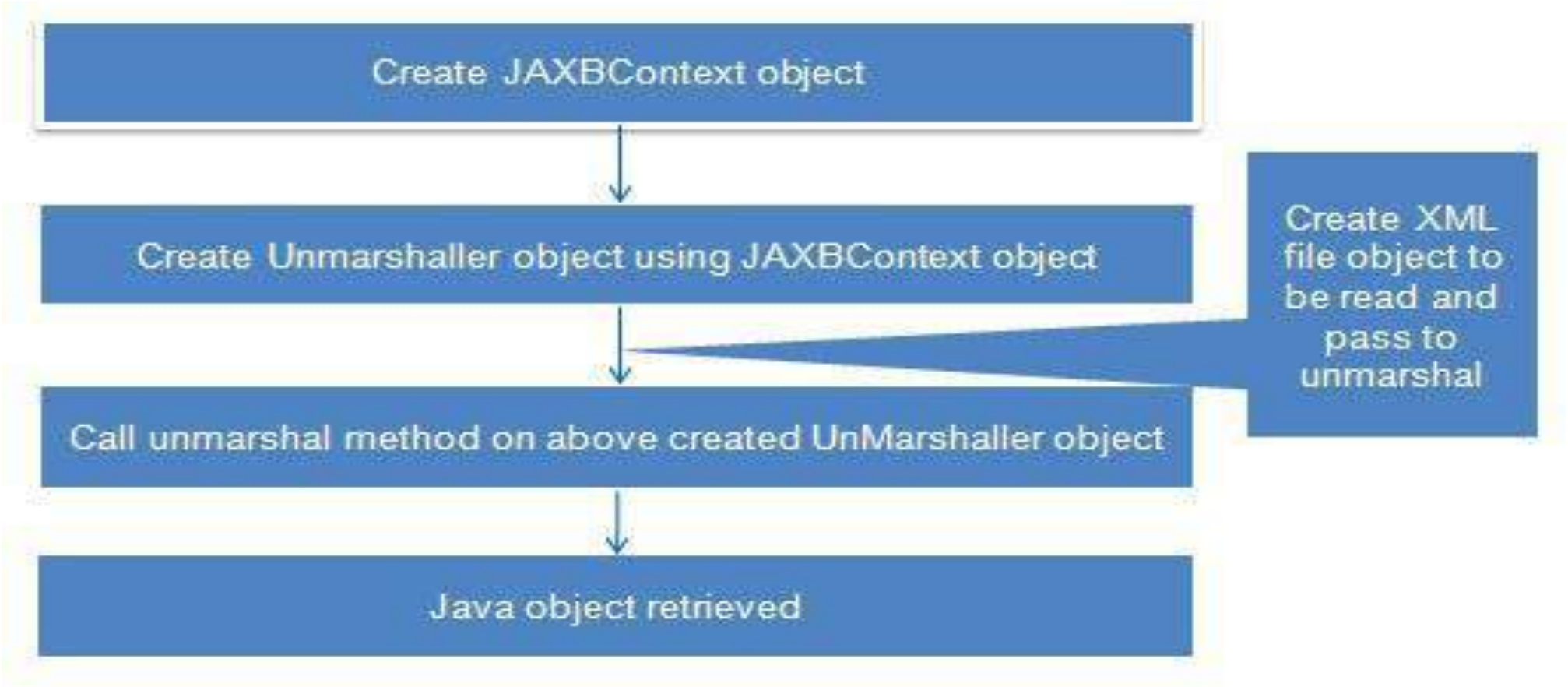
Using JAXB - Marshalling

```
// create JAXB context object
JAXBContext context = JAXBContext.newInstance(BookStore.class);

// create Marshaller
Marshaller m = context.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);

// Call marshall method
   // Write to System.out
   m.marshal(bookstore, System.out);
    // Write to File
   m.marshal(bookStore, new File("bookstore.xml"));
```

Pass main class

Instance of bookstore

Using JAXB - Unmarshalling



Using JAXB - Unmarshalling

```
// create JAXB context object
JAXBContext context = JAXBContext.newInstance(BookStore.class);

// create Unmarshaller
Unmarshaller um = context.createUnmarshaller();

// Call ummarshall method
BookStore bookStore2 =
   (BookStore) um.unmarshal(new FileReader("bookstore.xml"));
```

Pass main class

Outline – XML B - JAXB

- What is JAXB?
- Marshal and Unmarshal
- Annotating classes
- Generating classes from XSD

JAXB Annotations

- JAXB uses annotations to indicate the central elements which should be expressed as xml elements
- An **annotation**
  - is a form of syntactic metadata that is added to Java source code.
  - is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.
- JAXB annotations are executed during runtime.

```
@AnnotationKeyword(parameter1, parameter2, …)
```

Example Bookstore

- The given class `BookStore.java` contains an address (`Address.java`) and an `ArrayList` of books (`Book.java`).
- Annotate the classes `BookStore.java` and `Book.java` with JAXB annotations to persist the bookstore as XML.

- Given classes:
  - `Bookstore.java`
  - `Book.java`
  - `Address.java`
  - `BookMain.java`

JAXB Annotation – `@XmlRootElement`

```
@XmlRootElement(name = "newName", namespace = "namespace")
```

- Defines the root element for an XML tree, **the one** that can function as an XML document, should be annotated with `XmlRootElement`. This annotation corresponds to an `xsd:element` construct being used at the outermost level of an XML schema
- Its optional elements are `name` and `namespace`. By default, the class name is used as name.
  - Use the element `name` to change the name of the tag
  - Use the element `namespace` to add a namespace to this element
- Per default: all simple fields become sub elements (by reflection)
- Example: **`BookStore.java`**

```
@XmlRootElement(name = "store", namespace = "http://ti.ch.bfh/oop2/bookstore")
```

- **Attention**: Class must have a no-arg default constructor. (Nonexistence -> CPE!!)

JAXB Annotation – `@XmlElement`

```
@XmlElement(name = "newName")
```

- Define the XML element which will be used.
- Adds information, which isn't part of a Java class declaration.
- It permits to define the XML element **name**, the **namespace**, whether it is **optional** or **nillable** (can contain null value), a **default value** and the **Java class**.
- Also allows to turn **methods** into xml elements, e.g. BookStore.getNmbOfBooks()

- Example: **`BookStore.java`**

JAXB Annotation – `@XmlElementWrapper`

```
@XmlElementWrapper
```

- **Optional**
- Additional element for lists to distinguish between a list that is absent and an empty list.

- Example: **`BookStore.java`**

JAXB Annotation – `@XmlType`

```
@XmlType(propOrder = {"field2","field1",.. })
```

- **Optional**
- Adds information useful for an xml type, which isn't part of a Java class declaration.
- The `namespace` attribute provides the name of the target namespace.
- The attribute `propOrder` establishes an ordering of the sub-elements. (if not used an alphabetic order is used)

- Example: **`Book.java`**

- **Attention**: Class must have a no-arg default constructor

JAXB Annotation – `@XmlAttribute`

```
@XmlAttribute(name = "newName")
```

- **Optional**
- Adds information useful for an xml attribute, which isn't part of a Java class declaration.
- Can transform a field or method into an xml attribute

- Example: **`Book.java`**

Attention: Must be used together with **`@XmlAccessorType(XmlAccessType.FIELD)`** **!!**

JAXB Annotation – `@XmlAccessorType`

```
@XmlAccessorType(XmlAccessType.FIELD)
```

- **Optional**
- Controls how a class is serialized by default
- Default is `@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)`:
- `XMLAccesstype`:
  - `PUBLIC_MEMBER:` Every public getter/setter pair and every public field will be automatically bound to XML.
  - `FIELD:` Only the fields are bound to XML.
  - …

- Example: **`Book.java`**

JAXB Annotations

Annotation	Annotation for	Description
@XmlRootElement(name = "name" namespace = "namespace")	(Main) Class	Defines the root element for an XML tree
@XmlElement(name = "newName")	Field/Method	Define the XML element which will be used
@XmlElementWrapper	Field/Method	Additional element for lists to distinguish between a list that is absent and an empty list.
@XmlType(propOrder = { "field2", "field1",.. })	Class	Allows to add more infos to a class, e.g. the order of fields
@XmlSeeAlso({class1, class2, …})	Class	Allows to use class hierachies with correct type information
@XmlAttribute(name = "newName")	Field/Method	Defines an attribute
@XmlAccessorType(XmlAccessType.FIELD)	Class	Controls how a class is serialized by default
…		

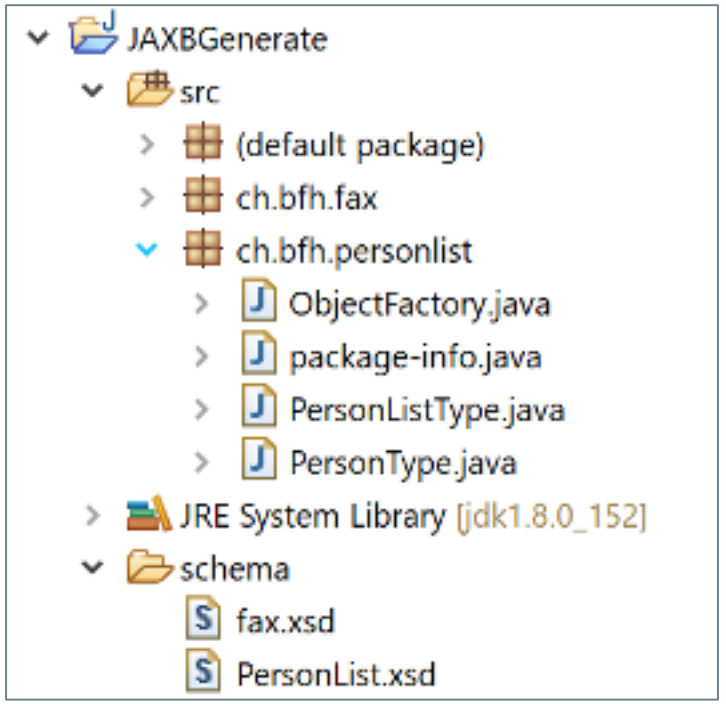
Online

- Installation
- Marshal and Unmarshal
- Annotating classes
- Generating classes from XSD

Create Java Classes from XML Schema 1/5

- From the XSD file: Generate – Generate JAXB classes…

- Example: PersonList.xsd
  - Generated classes



- Let’s have a look in the generated code

Create Java Classes from XML Schema 2/5

The generated code

- **package-info.java**
  - Generated package descriptor for JavaDoc
  - Documents the binding of the schema to the package

- **PersonType.java**
  - simple XMLType (`@XMLType`)
  - offers getter and setter for the field *firstname*

```
<xsd:complexType name="PersonType">
    <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
```

Create Java Classes from XML Schema 3/5

- **PersonListType.java (V1)**
  - Flagged as `@XmlRootElement`
  - simple XMLType (`@XMLType`)
  - offers getter and setter for the field *person* (only one object of type `PersonType`)

```
<xsd:element name="PersonList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="person" type="PersonType"
                maxOccurs="1" minOccurs="1" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Create Java Classes from XML Schema 4/5

- **PersonsType.java (V2)**

```
<xsd:element name="PersonList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="person" type="PersonType"
                maxOccurs="unbounded" minOccurs="1" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

- `getPerson()` returns now a List of PersonType (`List<PersonType>`)

Create Java Classes from XML Schema 5/5

- **ObjectFactory.java**
  - It's a factory !!!
  - Instead to create an object while calling the constructor ..

```
SomeObject o = new SomeObject();
```

  - A static method is used

```
SomeObject o = SomeObjectFactory.createNewInstance();
```

Marshalling the generated Java Classes

- Example: **MarshallPersons.java**
- Produces the following result

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:PersonList xmlns:ns2="http://www.bfh.ch/personList/">
    <person>
        <firstName>Adam</firstName>
        <lastName>Meier</lastName>
    </person>
    <person>
        <firstName>Eva</firstName>
        <lastName>Schulze</lastName>
    </person>
</ns2:PersonList>
```

Validate and Unmarshall the generated XML

- Example: **UnMarshallPersons.java**
- To validate a XML file against a XSD

```
public static boolean validateXmlSchema(String xsdFile, String xmlFile) {
   boolean result = true;
   try {
     SchemaFactory factory = SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
     Schema schema = factory.newSchema(new File(xsdFile));
     Validator validator = schema.newValidator();
     validator.validate(new StreamSource(new File(xmlFile)));
   } catch (IOException | SAXException e) {
     System.out.println("Exception: " + e.getMessage());
     result = false;
   }
   return result;
 }
```

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

JavaFX FXML & MVC

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS2018

- Computer science

Outline – JavaFX FXML & MVC

- Separation of layout and logic
- Observer Design Pattern
- MVC Design Pattern

Outline – JavaFX FXML & MVC

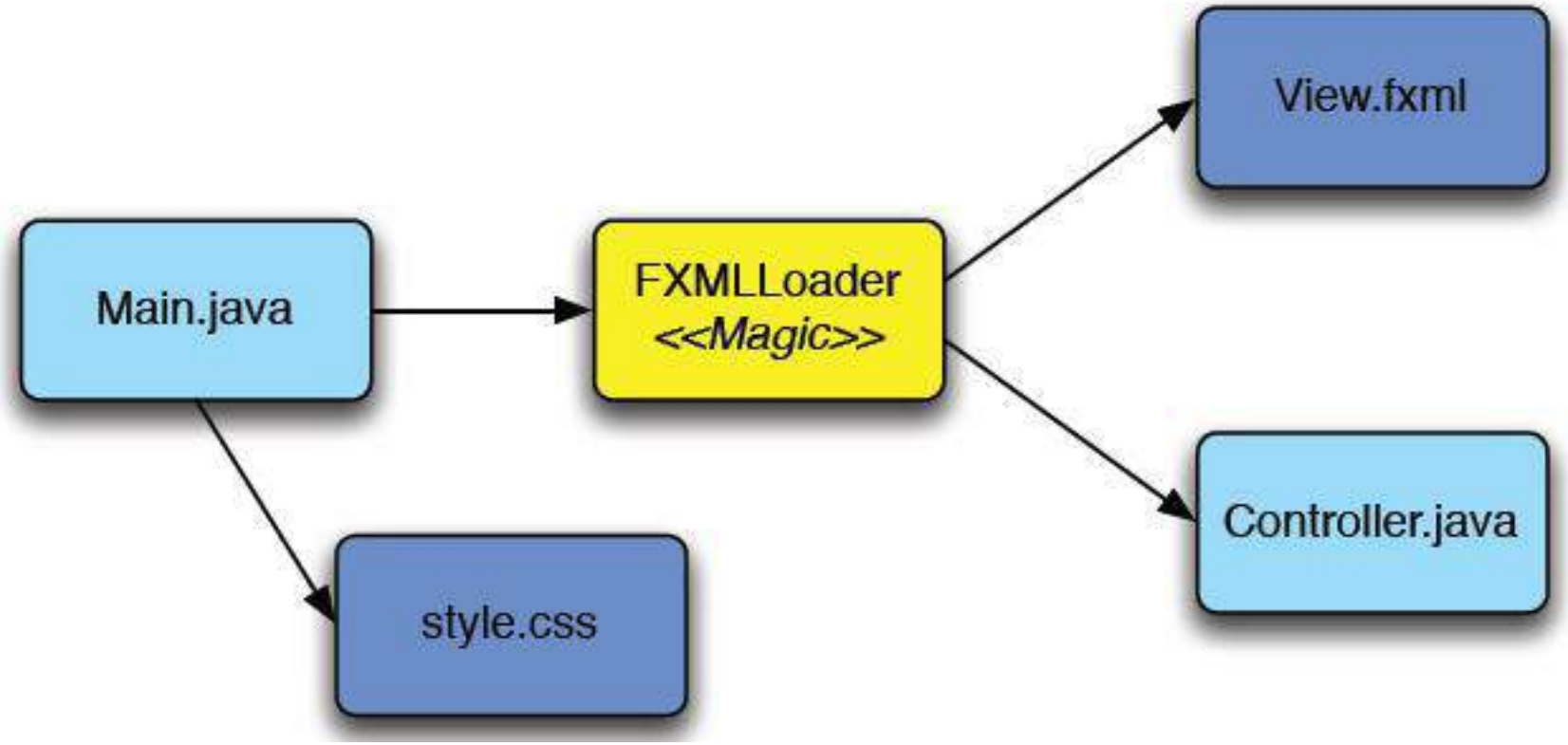
- Separation of layout and logic
- Observer Design Pattern
- MVC Design Pattern

Until now

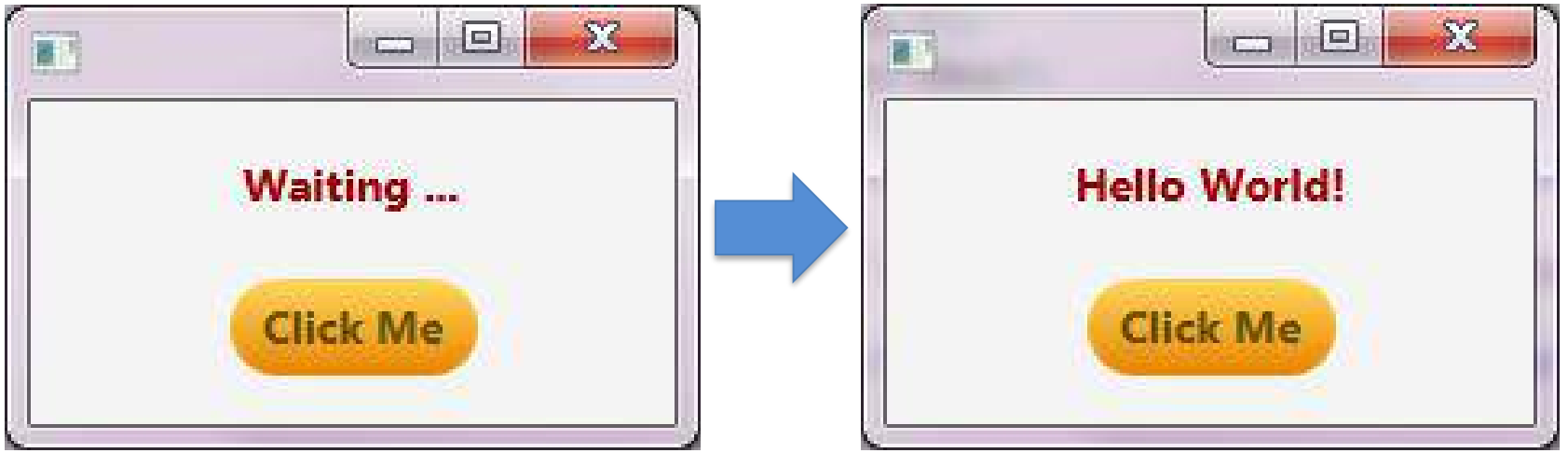
```
public class LabelDemo extends Application {

   public void start(Stage primaryStage) throws Exception {
      final TilePane pane = new TilePane();
      pane.getChildren().add(new Label("I'm a Label"));
      pane.getChildren().add(new Label("I'm another Label"));
      primaryStage.setScene(new Scene(pane, 400, 50));
      primaryStage.setTitle("Label Demo");
      primaryStage.setResizable(true);
      primaryStage.show();
   }
   public static void main(String[] args) {
      launch(args);
   }
}
```

UI elements are explicitly defined

Separation of layout and logic



ClickMe Example



Package clickme

- *Main.java*
- *SimpleController.java*
- *View.fxml*
- *application.css*

Main.java

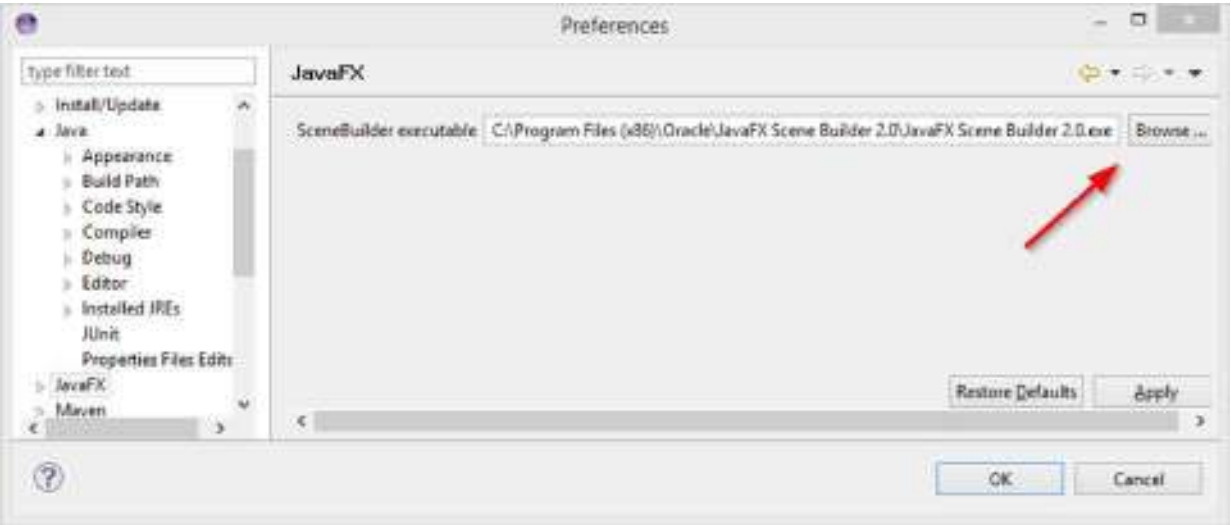
```
public class Main extends Application {
   public void start(Stage primaryStage) {
     try {
       FXMLLoader loader = new FXMLLoader(getClass().getResource("View.fxml"));
       Parent root = (Parent) loader.load();
       Scene scene = new Scene(root);
       primaryStage.setScene(scene);
       primaryStage.show();
    } catch(Exception e) {
       e.printStackTrace();
  }}

   public static void main(String[] args) {
      launch(args);
   }}
```

Create an FXMLLoader instance based on the fxml file

Create the node hierarchy by calling method `load()`

JavaFX with SceneBuilder

- Scene Builder needs to be installed separately and configured in Eclipse
  - Download: http://www.oracle.com/technetwork/java/javafxscenebuilder-1x-archive-2199384.html#javafx-scenebuilder-2.0-oth-JPR
  - Configure: Navigate to the *JavaFX* preferences. Specify the path to your Scene Builder executable.



- Complete Tutorial: http://docs.oracle.com/javafx/scenebuilder/1/use_java_ides/sb-with-eclipse.htm

View.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.text.*?>
<?import java.lang.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns=http://javafx.com/javafx/8  xmlns:fx="http://javafx.com/fxml/1"
      alignment="CENTER" prefHeight="100.0" prefWidth="200.0"
      fx:controller="Clickme.SimpleController">
   <children>
      <Text fx:id="message" text="Waiting ..." textAlignment="CENTER"/>
      <Button alignment="CENTER" defaultButton="true"
         onAction="#handleClickMe"
         text="Click Me">
   </children>
</VBox>
```

Imports

Default Namespace

fxml Namespace

Controller class

Name of the element

Event Handler

SimpleController.java

```
public class SimpleController {

    @FXML
    private Text message;

    @FXML
    protected void handleClickMe(ActionEvent event) {
      this.message.setText("Hello World!");
    }
}
```

Annotation

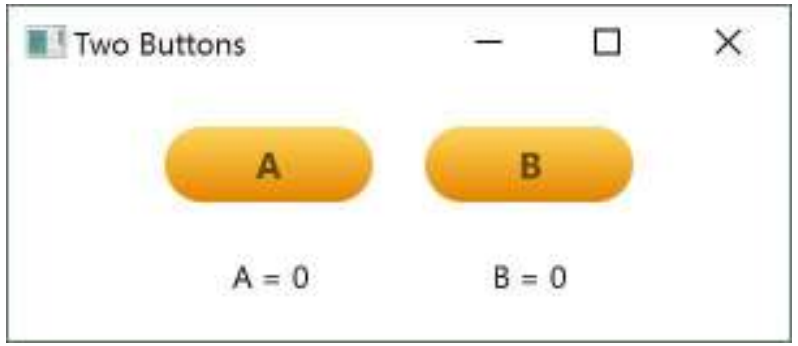
To use an UI element, declare it as private field with the fx:id as name and the type of the UI element

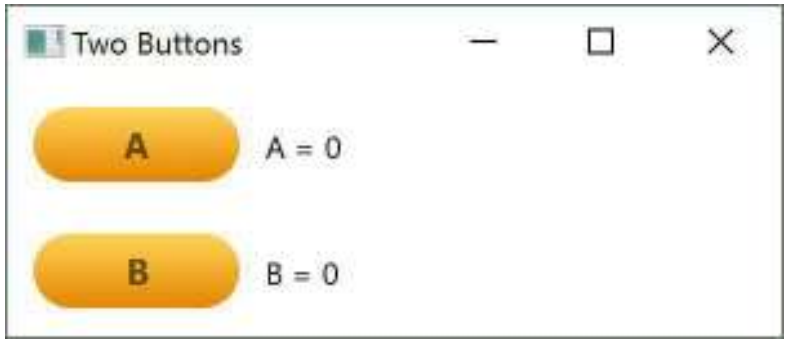
Define a method with an parameter ActionEvent as event handler

Exercise

Write a JavaFX application with two buttons A and B. Each time Button A respectively Button B is clicked, a counter is increased and the number of clicks is displayed (separately for Button A and B).

Separate the Logic from the Layout! Provide two different UI descriptions (View.fxml) !

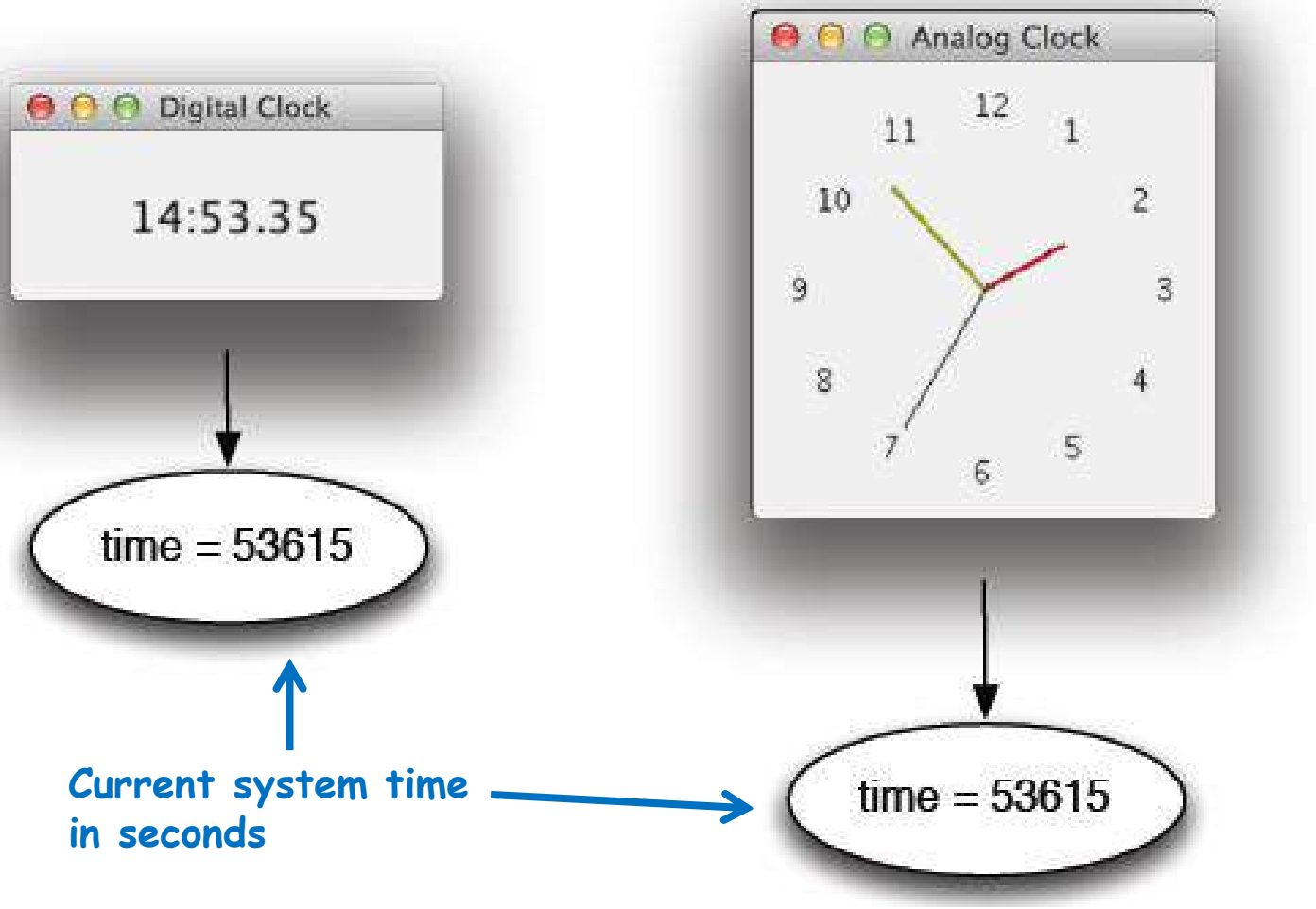




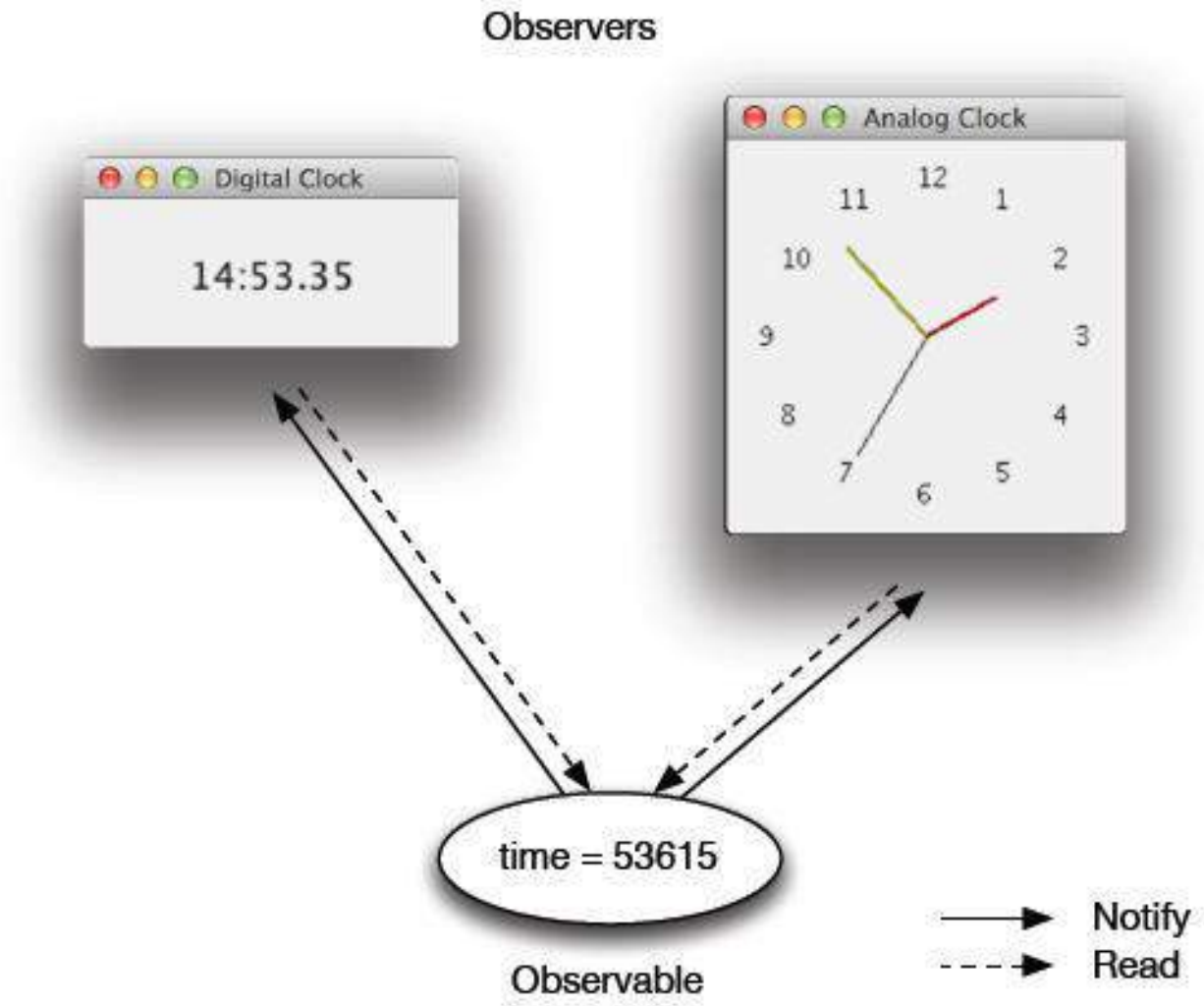
Outline – JavaFX FXML & MVC

- Separation of layout and logic
- Observer Design Pattern
- MVC Design Pattern

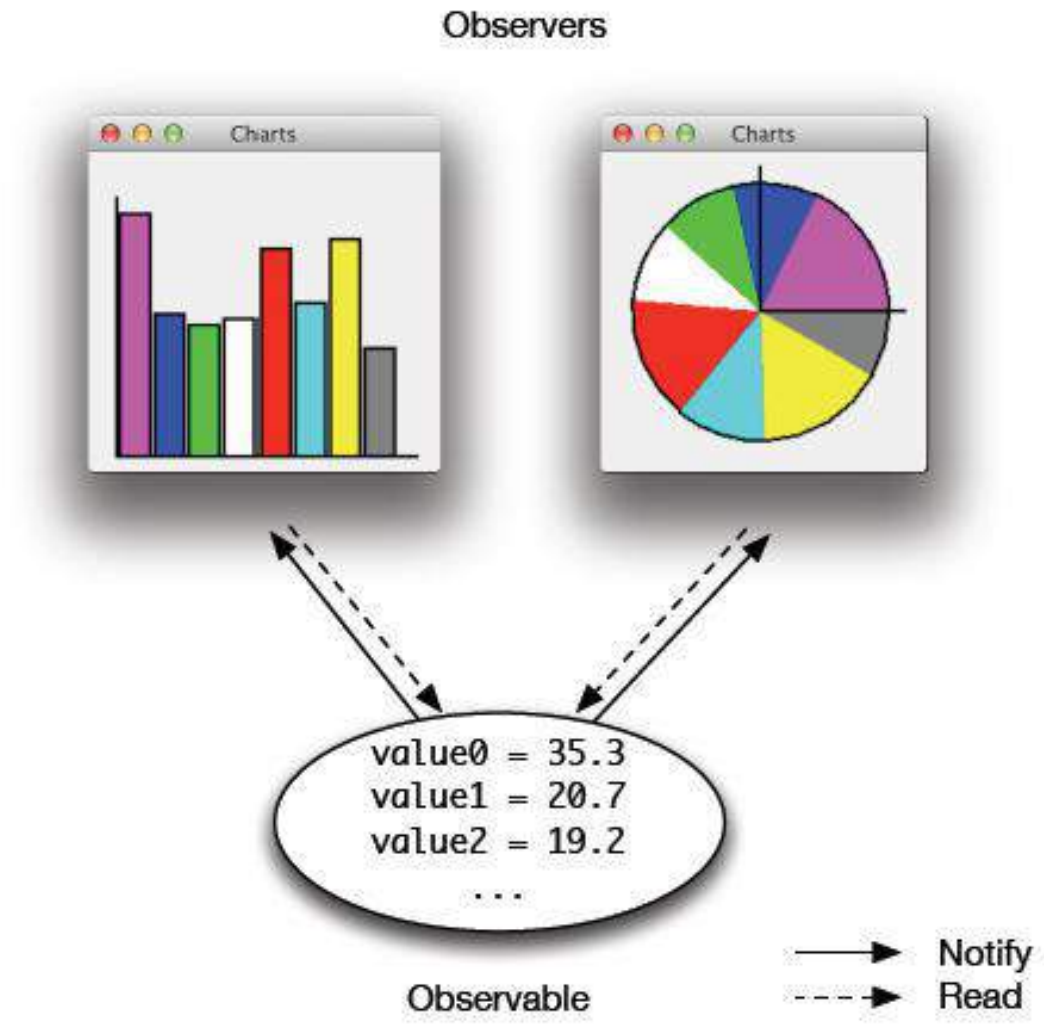
Observer Design Pattern



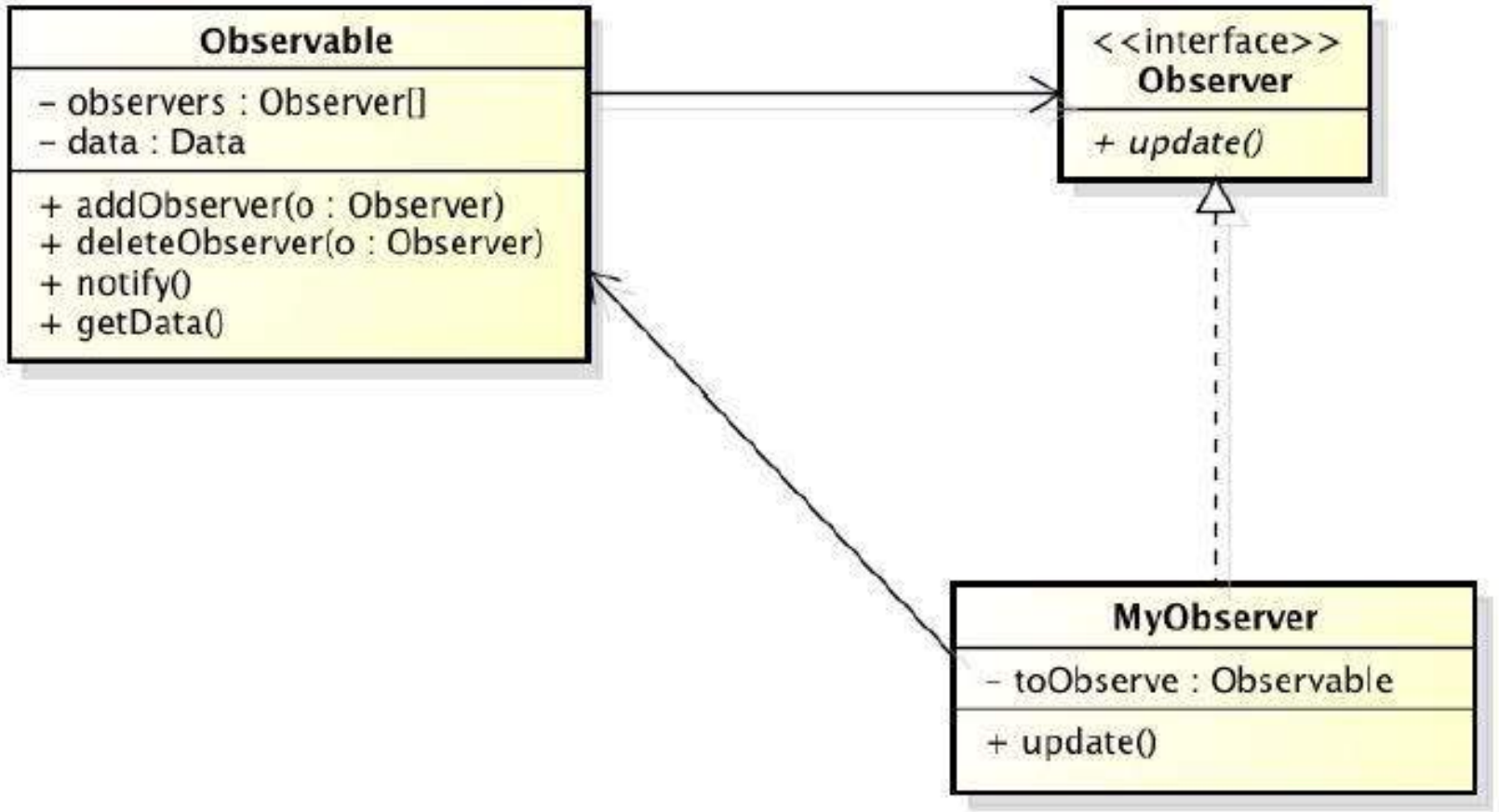
Observer Design Pattern



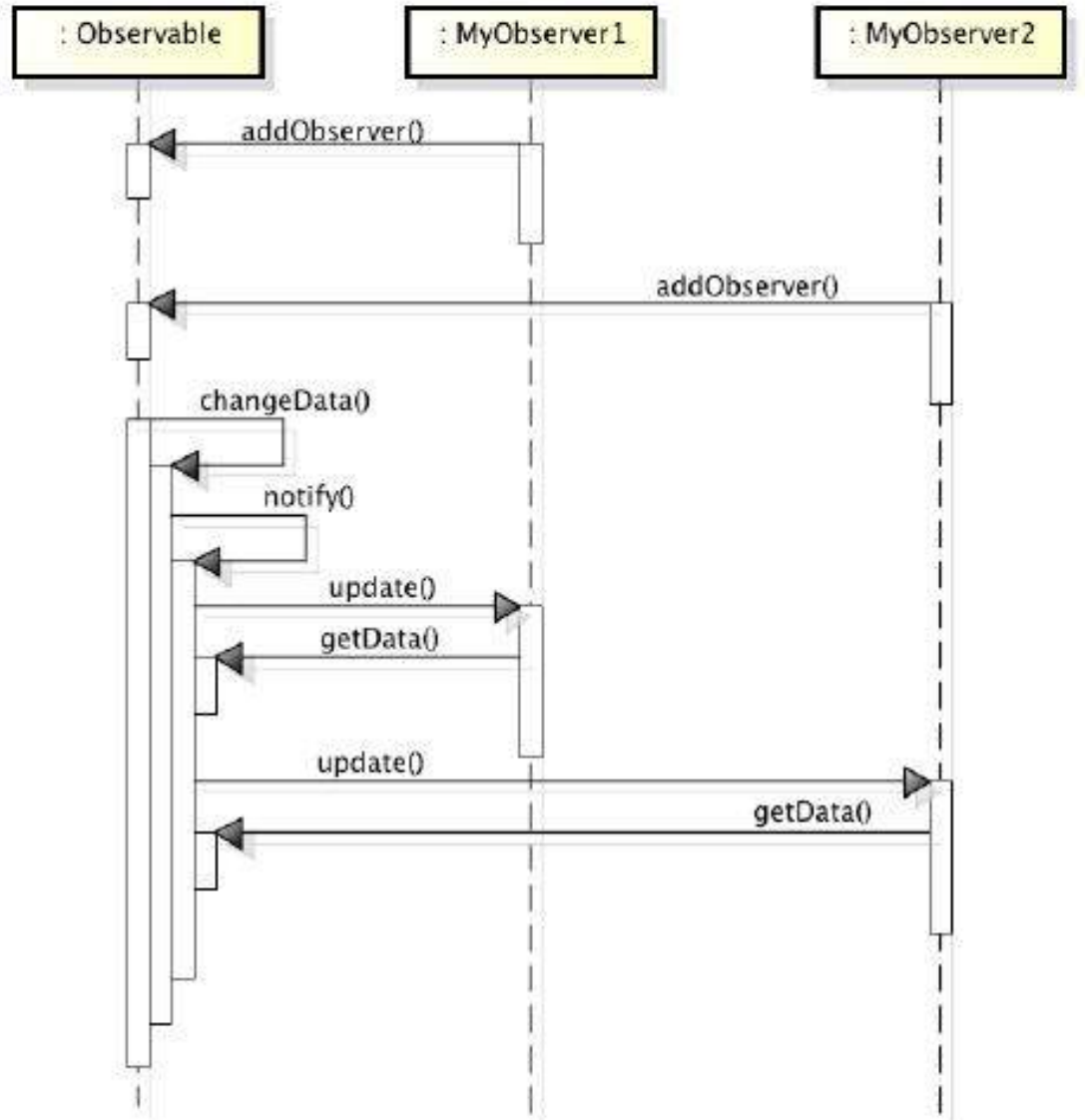
Observer Design Pattern



Observer Class diagramm



Observer Sequence diagram



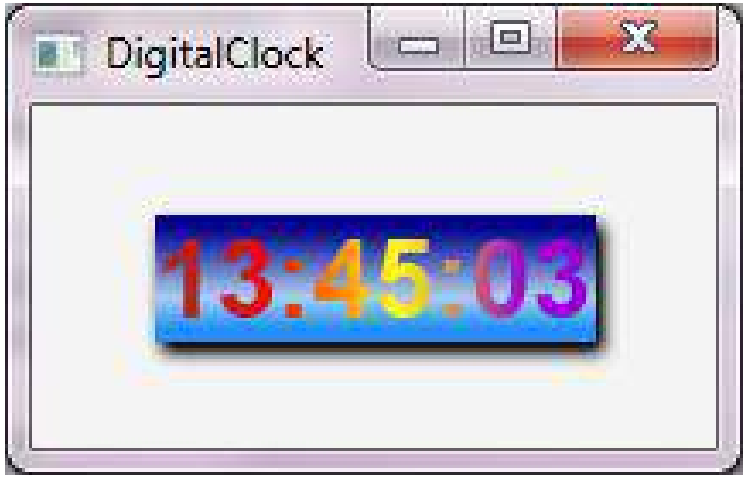
Observer Design Pattern in Java

- The Observer Design Pattern is very easy to implement in Java

- Java supports the functionality of the pattern with the following 2 classes:
  - **`java.util.Observable`** - base class for *Observables*
  - **`java.util.Observer`** - Interface for *Observer*

- A concrete *Observable* has to call only the methods `setChanged()` and `notifyObservers()` after a change of its internals
- The base class *Observable* realizes automatically all necessary functionalities.

Example DigitalClock



Package digitalclock

- *Main.java*
- *DigitalClock.java*
- *Time.java*

Observable Time.java 1/2

```
import java.util.Observable;
public class Time extends Observable implements Runnable {
   private long time;

   public Time() {
      this.time =System.currentTimeMillis();
      Thread t = new Thread(this);
      t.setDaemon(true);
      t.start();}

   public void run() {
      while (true) {
        try {
          Thread.sleep(1000);
          this.increaseTime();
          } catch (InterruptedException e) {}
      }}
```

Initialize the object and create a Thread

Update the time every second

Observable Time.java 2/2

```
    private void increaseTime() {
        this.time =System.currentTimeMillis();
        // Important:
        // Call setChanged() before calling notifyObservers()
        this.setChanged();
        this.notifyObservers();
    }

    public long getTime() {
         return this.time;
    }
}
```

Inform the observers

Used by the observers the get the latest values Corresponds to the `getData()` method

Observer DigitalClock

```
import java.util.Observable;

public class DigitalClock extends Label implements Observer {
  private final SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss a");

  public DigitalClock(final Time time) {
     time.addObserver(this);
  }

  public void update(Observable o, Object arg) {
   // Make sure, the GUI is updated in the JavaFX Application Thread!
   Time time = (Time) o;
   Platform.runLater(() -> {
        this.setText(sdf.format(time.getTime()));   });
  }
}
```

Time formatter

Register the observer

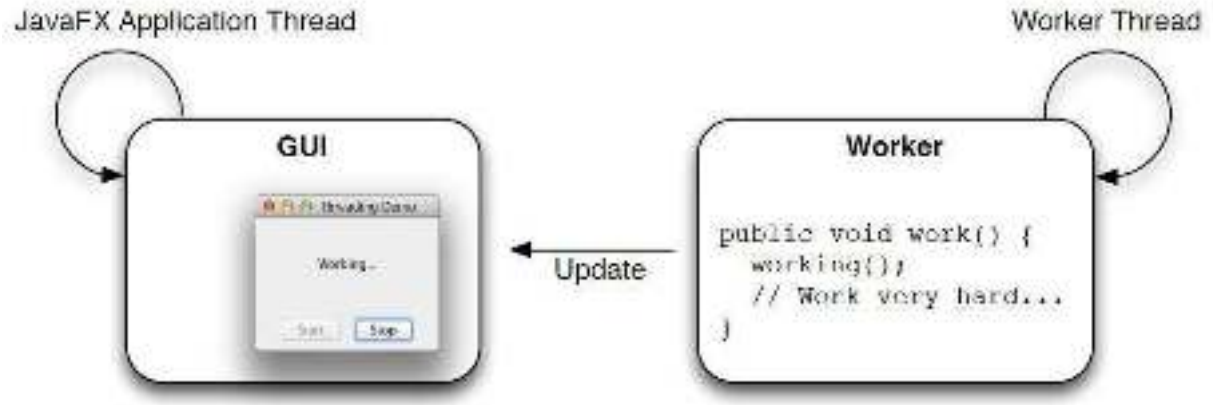
Update() method is called
when Observable is changed

Cast the observable

Pull the data

Thread-safety

- **Der Scene Graph von JavaFX ist nicht thread-safe!**
  - Der Scene Graph und somit das GUI darf nur vom JavaFX Application Thread verändert und aktualisiert werden
  - Wird der Scene Graph von einem anderen Thread verändert, wird eine Exception geworfen
  - Mittels `Platform.runLater()` können Ereignisse in die Event-Queue geschrieben werden, welche anschliessend vom UI Thread verarbeitet werden



Main.java

```
public class Main extends Application {

   public void start(Stage primaryStage) throws Exception {
      VBox pane = new VBox(5);
      Time time = new Time();
      DigitalClock clock = new DigitalClock(time);

      pane.getChildren().add(clock);
      primaryStage.setScene(new Scene(pane, 200, 100));
      primaryStage.show();
   }

    public static void main(String[] args) {
      launch(args);
    }
}
```

Observable

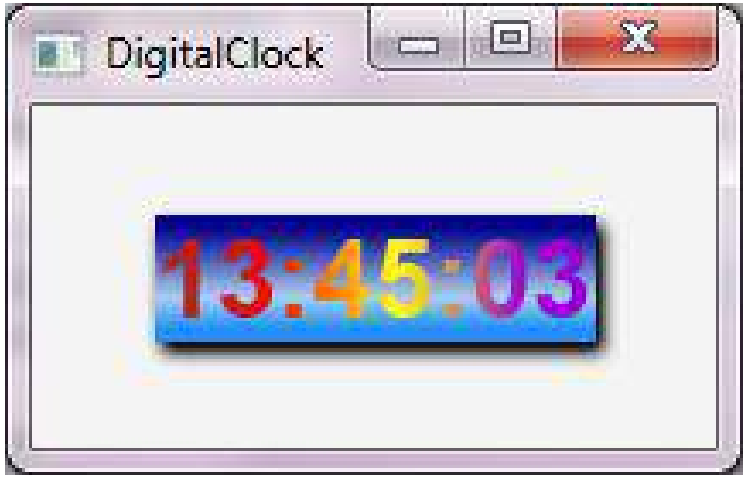
Observer

Pull or Push

- **Pull**
  - The observer gets the changed data explicit form the observable
  - Ex. `DigitalClock` calls `time.getTime()`

- **Push**
  - The observable hands over the changed data with the update request.
  - A **Data Transfer Object** is used, e.g. the `TimeObject`

Example DigitalClock (push)



Package digitalclockpush

- *Main.java*
- *DigitalClock.java*
- *Time.java*
- *TimeObject.java*

Data Transfer Object

- Stores data which are transferred from the observable to the observer
- Is immutable. (After creation the data can only be read).

```
public class TimeObject {
    private long time;

    public TimeObject(long time) {
        this.time = time;
    }

    public long getTime() {
        return this.time;
    }
 }
```

Data Transfer Object

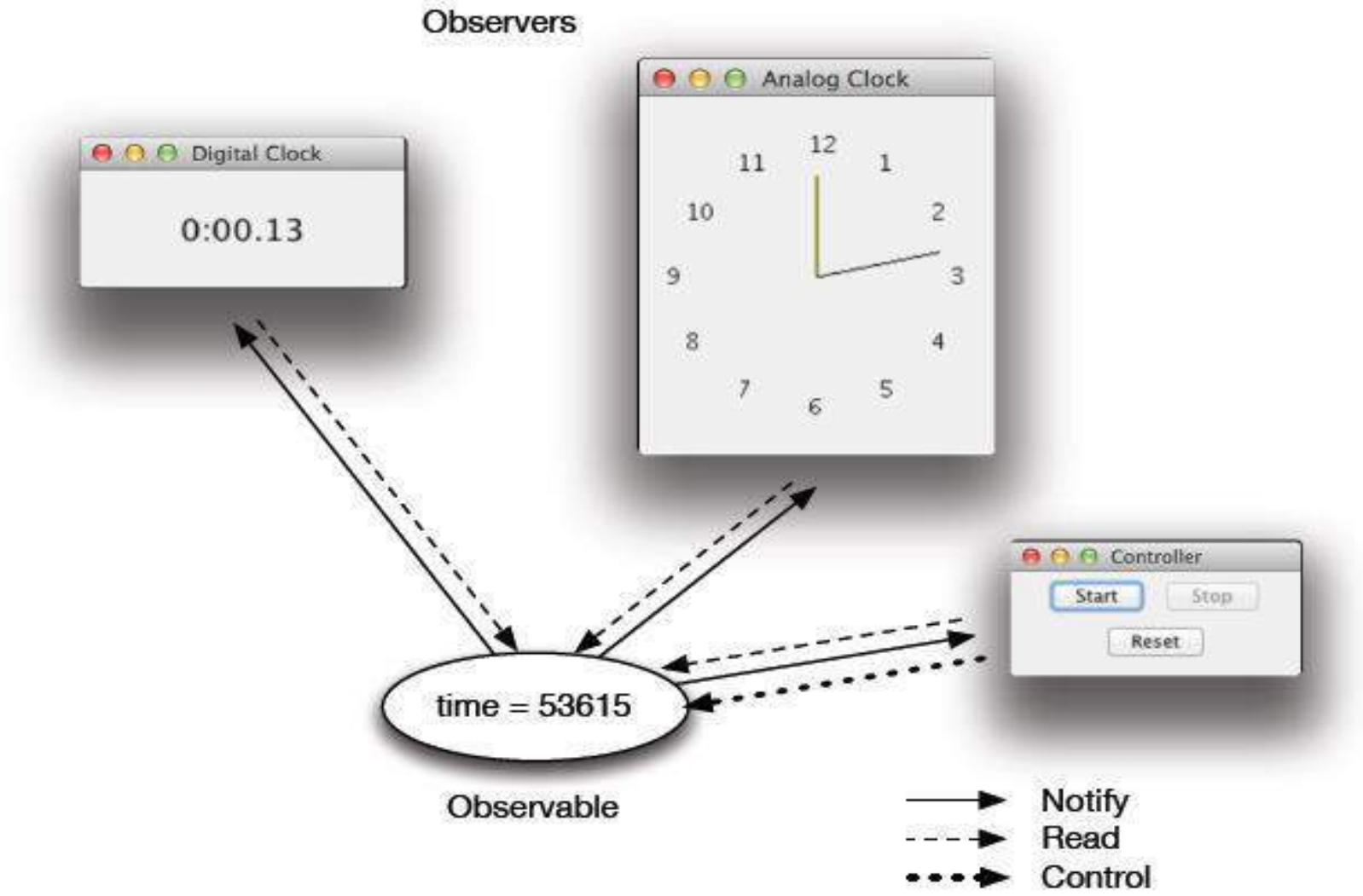
```
public class Time extends Observable implements Runnable {
...
    private void increaseTime() {
        this.time =System.currentTimeMillis();
        this.setChanged();
        this.notifyObservers(new TimeObject(this.time));   }
...
}
public class DigitalClock extends Label implements Observer {
...
    public void update(Observable o, Object arg) {
       TimeObject timeObject = (TimeObject) arg;
       Platform.runLater(() -> {
            this.setText(sdf.format(timeObject.getTime()));
        });
...
}
```

Cast the argument

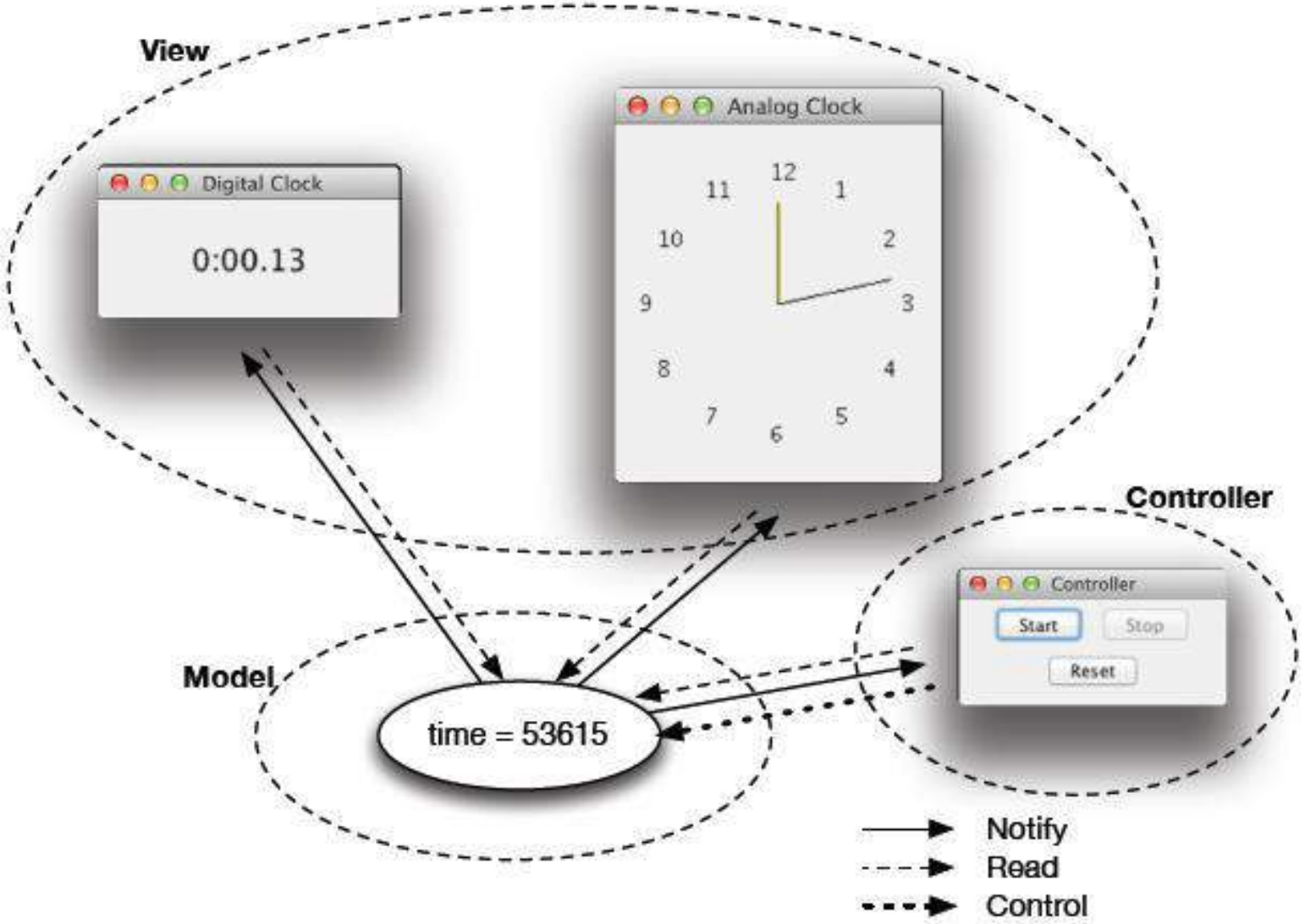
Outline – JavaFX FXML & MVC

- Separation of layout and logic
- Observer Design Pattern
- MVC Design Pattern

MVC Pattern


Observers
Digital Clock
0:00.13
Analog Clock
12
1
2
3
4
5
6
7
8
9
10
11
Controller
Start
Stop
Reset
time = 53615
Observable
Notify
Read
Control

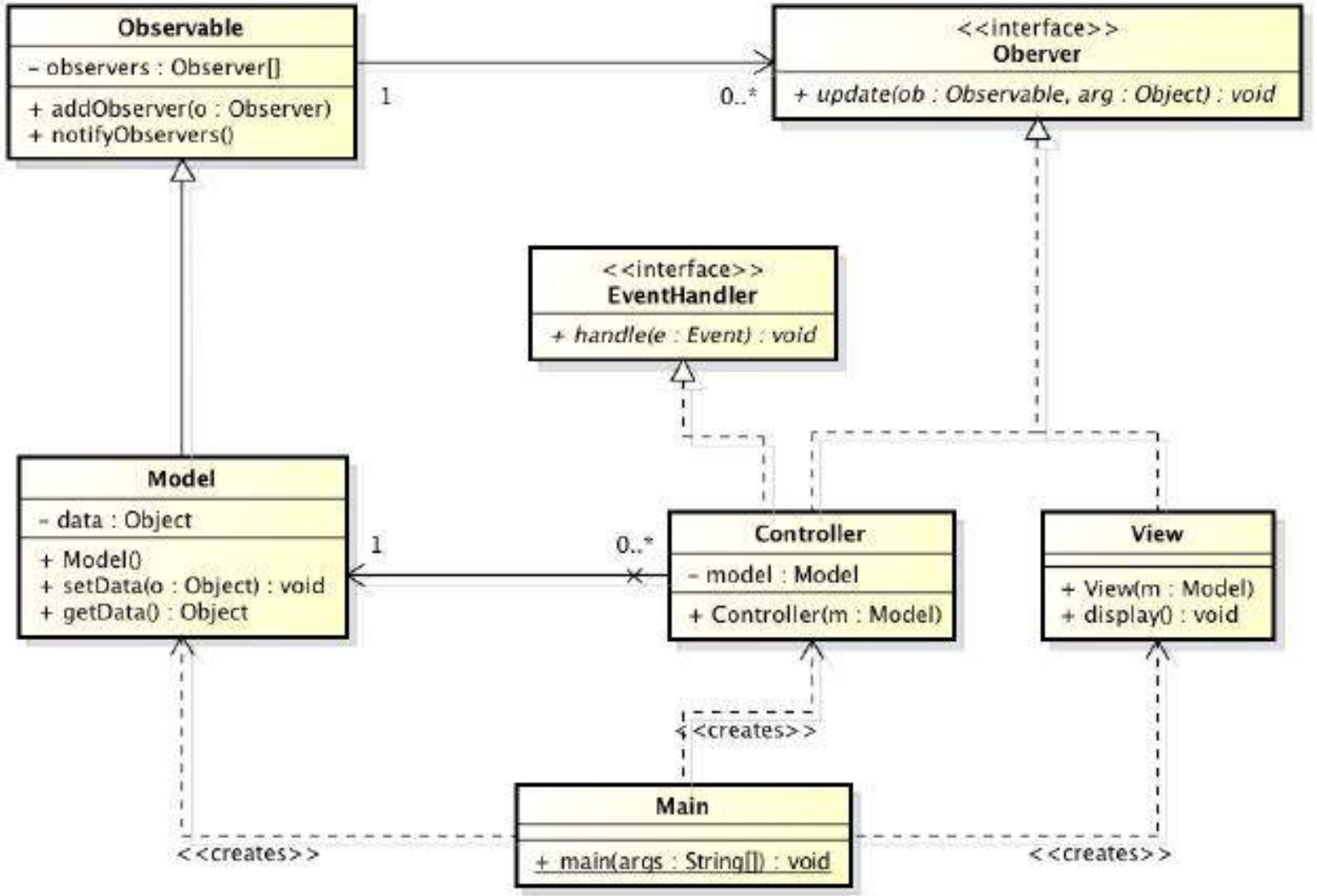
MVC Pattern


View
Digital Clock
0:00.13
Analog Clock
12
1
2
3
4
5
6
7
8
9
10
11
Controller
Controller
Start
Stop
Reset
Model
time = 53615
Notify
Read
Control

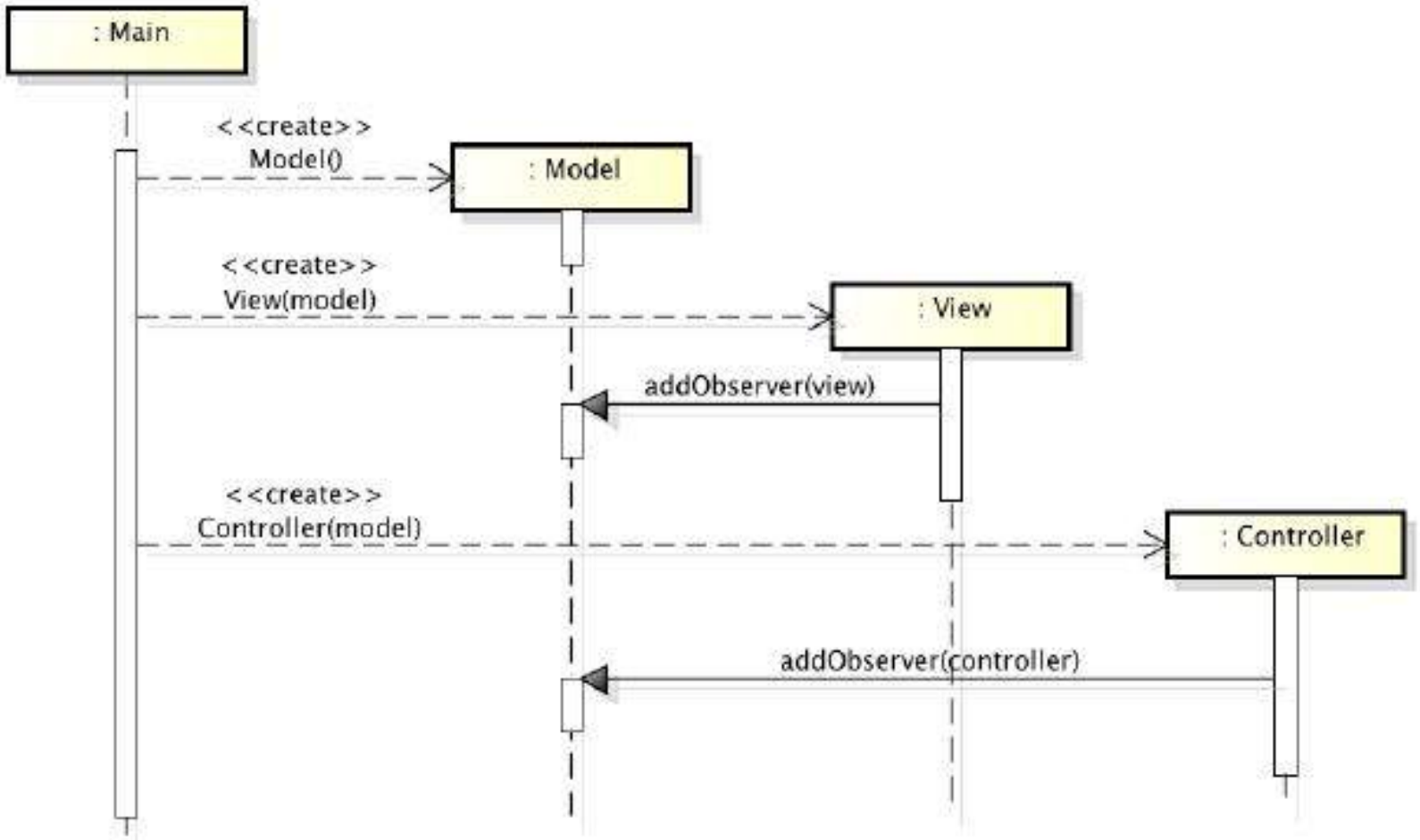
MVC Pattern

- **Model**
  - Data and data processing
- **View**
  - Presentation of the data
- **Controller**
  - User Input
- Goal: **Decoupling** of the different parts of the application

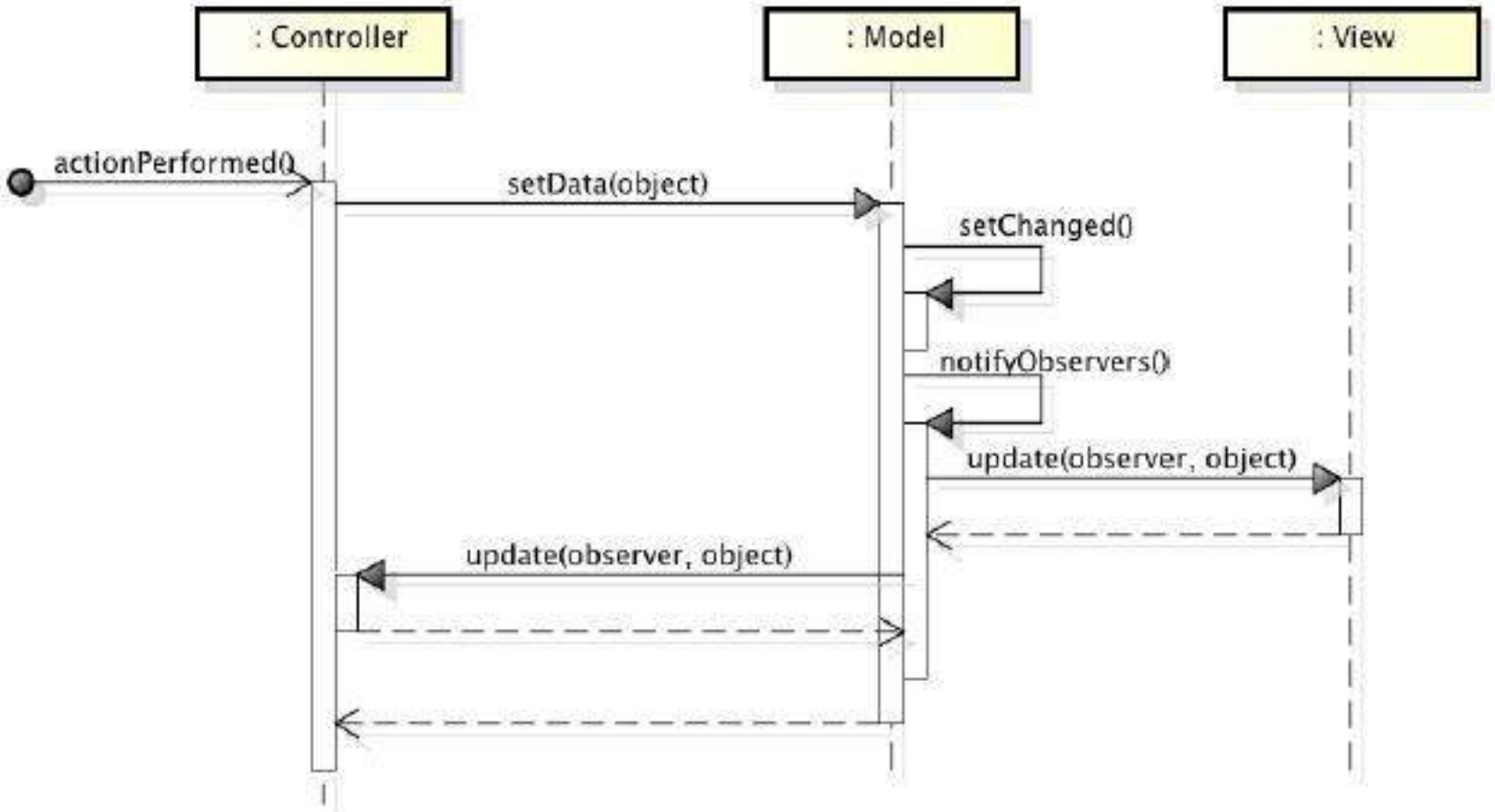
MVC class diagram



MVC Initialization – sequence diagram



MVC event handling – sequence diagram



Example StopWatch MVC

Package stopwatch_mvc

- *Main.java*
- *Controller.java*
- *Timer.java*
- *Stopwatch.java*
- *Stopwatch_blue.fxml / Stopwatch_yellow.fxml*
- *blue.css / yellow.css*

MVC with JavaFX

- How to get to the Model into the controller?
  - static methods
  - Model is implemented as Singleton
  - Model is created and initialized in the `Main` and passed to the `Controller`
  - In bigger projects, it can be reasonable to use
    - Dependency Injection (http://martinfowler.com/articles/injection.html)
    - Event Bus (https://code.google.com/p/guava-libraries/wiki/EventBusExplained)

MVC with JavaFX

Example `Main`: Model is created and initialized in the `Main`

- Variant 1: pass the model to the Controller using an `init` method

```
Model model = new Model();
// Create an FXMLLoader instance based on the FXML
FXMLLoader loader = new FXMLLoader(
  getClass().getResource("View.fxml"));

// Create the node hierarchy by calling load
Parent root = (Parent) loader.load();

// Pass the model to the controller by calling init
loader.<Controller>getController().init(model);
```

Pass the model

MVC with JavaFX

- Variant 2: pass the Model to the Controller using the constructor

```
Model model = new Model();
 // Create an FXMLLoader instance based on the FXML
 FXMLLoader loader = new FXMLLoader(
   getClass().getResource("View.fxml"));

 // Set Controller explicitly (fx:controller must be
 // removed from FXML!)
 loader.setController(new Controller(model));
  Parent root = (Parent) loader.load();
```

Pass the model

MVC with JavaFX

Example `Controller`: Model is passed using an `init` method

```
public class Controller implements Observer {
  private Model model;
  @FXML private Label label;

  // Called by FXMLLoader after the node hierarchy has been created
  @FXML protected void initialize() {
   this.label.set...   }

  // Called by the Main after initialize() has been called
  public void init(Model model) {
     this.model = model;
    this.model.addObserver(this);
     this.label.setText(this.model.get... ); }

  @Override public void update(Observable o, Object arg) {
   // Update the gui in response to model changes (may via runLater())
    Platform.runLater(() -> { this.label.setText(this.model.get... );}); }
}
```

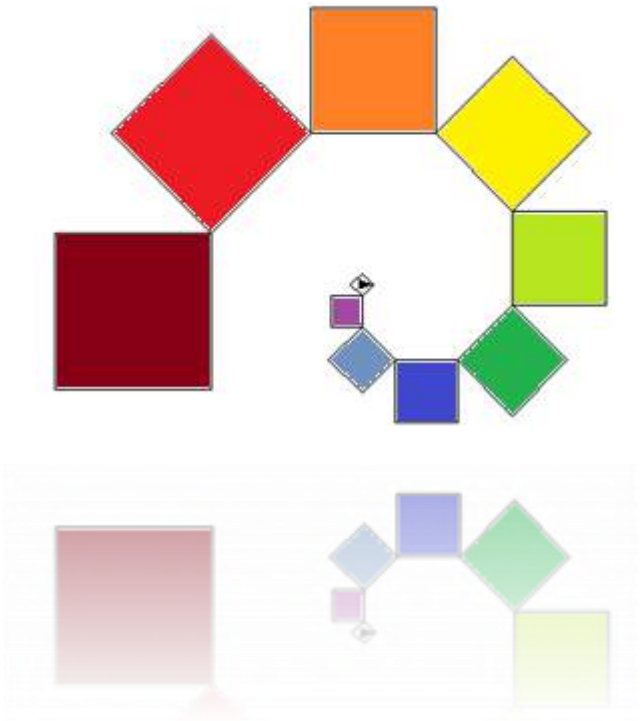
Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

Recursion

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS 18



Computer science

Algorithm schemes

- An **Algorithm scheme** is a process pattern or a general method with which a problem can be solved.
- Not every scheme is equally suitable for every problem.
- Umso wichtiger ist es also, die verschiedenen Algorithmen-Schemata zu kennen.
- This makes it more important to know the different algorithm schemes:
  - Iteration
  - Recursion

Iteration

- A problem is solved by *iteration* if the associated algorithm uses a loop (`while` or `for` loop).
- Iteration makes sense, if data is stored in an array (or a list) and for each element the same operation is performed.

Example Iteration

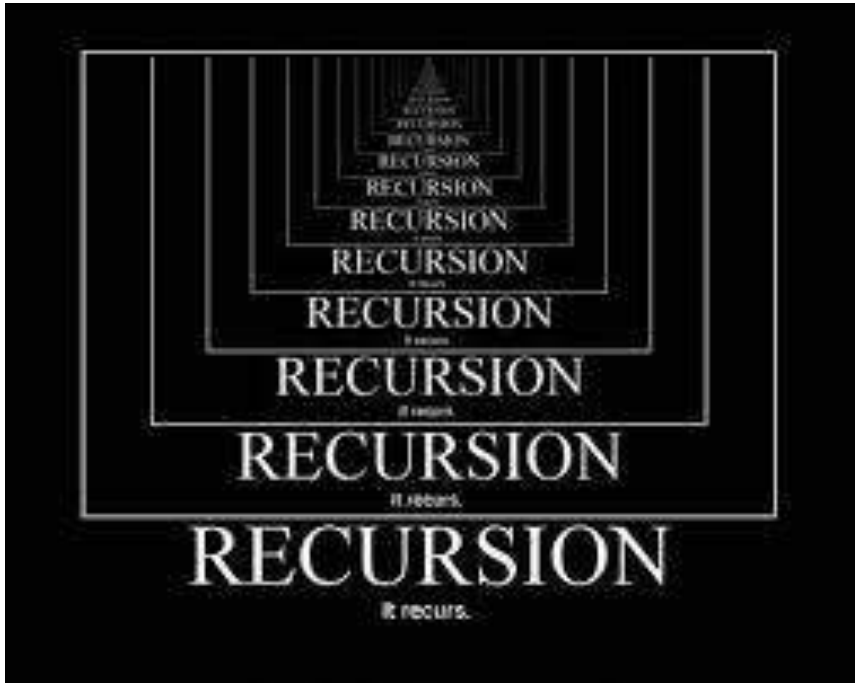
- Summation of 2 `Vector` objects

```
public static Vector<Double> sum(Vector<Double> v1, Vector<Double> v2)
      throws Exception
{
   if (v1.size() != v2.size()) {
     throw new Exception("Incompatible vector length");
   }
   Vector<Double> res = new Vector<Double>(v1.size());

   for (int i = 0; i < v1.size(); i++) {
     res.add(v1.get(i) + v2.get(i));
   }
   return res;
}
```

Recursion

Recursion is a fundamental concept of computer science.

- A method is called **recursive** if it calls itself directly or indirectly.
- Important: You must ensure that a termination condition exists so that the method always terminates.




```
public static long power(int n, int m)  {
   if (m == 0) {
     return 1;
   }
   return n * power(n, m--);
 }
```

Divide and Conquer

- **Divide and conquer** is a powerful algorithm design paradigm
  - Break down a problem recursively into two or more sub-problems of the same (or related) type
  - Continue until the problem becomes simple enough to be solved directly
  - Merge the solutions of the sub-problems to get a solution for the original problem

- Benefits of divide and conquer
  - Powerful tool for solving conceptually difficult problems
  - Often provides a natural way to design efficient algorithms
  - Sub-problems can be executed in parallel

- **Decrease and conquer** is a variation of divide and conquer, where the problem is simplified into a single sub-problem

Recursive Algorithm

- **Recursive Algorithm**: Algorithm that uses itself as part of the solution
- **Recursive Call**: A method call in which the method being called is the same as the one making the call
- **Direct Recursion**: Recursion in which a method directly calls itself
- **Indirect Recursion**: Recursion in which a chain of two or more method calls returns to the method that originated the chain, e.g. A calls B, B calls C, and C calls A

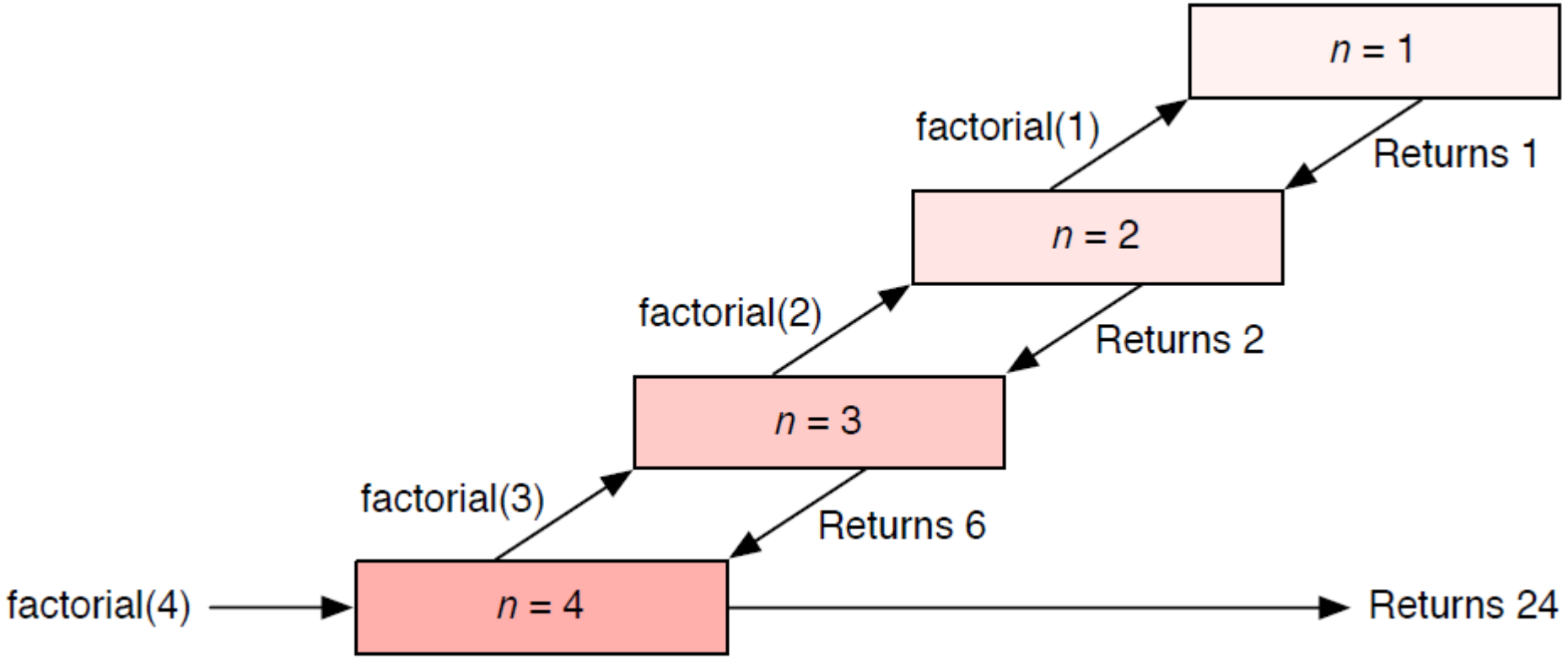
Example: Factorial (Recursion)

$$n! = \underbrace{1 * 2 * \cdots * n - 1}_{(n-1)!} * n = \begin{cases} 1 & n = 1 \\ n * (n-1)! & n > 1 \end{cases}$$

```
public static long factorial(int n) {
   if (n == 1) {
     return n;
   }
   return n * factorial(n - 1);
 }
```

Example: Factorial (Recursion)

The execution of a recursive algorithm uses a **call stack** to keep track of previous method call



Remarks

- Each recursion must have a **base case** to stop
- Each recursive call must reduce the problem size, i.e. leading inescapably to the base case
- A recursion is called **tail-recursive**, if the recursive call is always the last executed instruction
- Each recursive algorithm can be transformed into an iterative algorithm (and vice versa)
  - trivial for tail-recursions
  - not so easy in general (requires a **stack** to replace the callstack)
- In a recursive algorithm, the stack is hidden from the user

Example: Factorial (Iteration)

- The tail-recursive factorial algorithm can be replaced by a simple iterative loop

```
public static long factorial(int n) {
  int res = 1;
  while (n > 1) {
    res = res * n--;
  }
  return res;
}
```

Recursive vs. Iterative Algorithms

- Recursive algorithms are sometimes less efficient than their iterative counterparts
  - The overhead of managing the call stack is non-negligible
  - The space complexity is often O(n) or O(log n) instead of O(1)
  - Risk to get "stack overflow" errors
  - Some (functional) programming languages such as Scheme, Lisp, or Haskell detect tail-recursions automatically
- Nevertheless, recursion has been used to solve some of the biggest and hardest algorithmic problems in computer science !!!

Recursive vs. Iterative Algorithms

- Make sure that the number of recursive calls executed (the recursion depth) remains "reasonable", otherwise too much memory is used.
  - For example, only O(log(n)) recursive calls should be necessary when sorting n elements recursively.

- Recursion should be applied when the formulation of the solution becomes clearer and shorter.

- The effort (O) of the recursive solution must not be greater than the effort of the iterative solution.

Use «divide and conquer»

- **Divide**: Break down the problem until small problems that can be "easily" solved have been found.
- **Conquer**: Solve the smaller problems.
- **Merge**: Combine the partly solutions.

- ...can be used to solve conceptually difficult problems "simply".
- ...often helps to find efficient algorithms.
- Independent problems can be solved in parallel with Divide and Conquer.
- Since Java 7 parallelism in the language is comfortably supported.

Recursion and parallelism

- Recursive Computation of folder/file size

```
public static long sizeOf(File file) {
   long size = 0;
   if (file.isFile()) {
     size = file.length();
   } else {
     File[] children = file.listFiles();
     for (File child : children) {
       size += sizeOf(child);        //recursive call
     }
   }
   return size;
 }
```

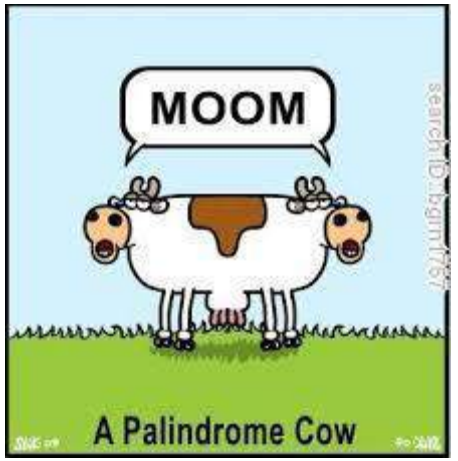
Recursion and parallelism

```
private static class FolderSizeParallel extends RecursiveTask<Long> {
    private final File root;

    public FolderSizeParallel(File file) {  this.root = file;  }

    protected Long compute() {
      if (root.isFile()) {  return root.length();   }
      List<FolderSizeParallel> tasks = new ArrayList<FolderSizeParallel>();
      for (File child : root.listFiles()) {
        FolderSizeParallel task = new FolderSizeParallel(child);
        task.fork();                                  divide problem
        tasks.add(task);
      }
      long size = 0;
      for (FolderSizeParallel task : tasks) {
        size += task.join();                          merge results
      }
      return size;
    }
  }
```

Example: Palindrome

- A Palindrome is a string that is equal to itself when you reverse all characters.
  - Ignore spaces, interpunction, and case (uppercase, lowercase).

- Example strings:
  - “A man, a plan, a canal–Panama!”
  - “Go hang a salami, I'm a lasagna hog”
  - “Madam, I'm Adam”
  - “Esope reste ici et se repose“
  - “Ein Neger mit Gazelle zagt im Regen nie“



- We now want to design a class that tests strings for being palindromes.

Palindrome: Designing the recursion

- Various ways to simplify inputs
  - Remove the first character
  - Remove the last character
  - Remove both the first and last characters
  - Remove a character from the middle
  - Cut the string into two halves

Palindrome: Recursive Computation

- “Reduce the problem to a simpler one of the same kind”.
  - Most promising simplification: remove first and last letters, …
    - “Madam, I'm Adam“ -> "adam, I'm Ada“. This string is a palindrome, too.
  - Thus, a word is a palindrome if
    - The first and last letters match, and
    - String obtained by removing the first and last letters is a palindrome.
  - What if first or last character is not a letter? Ignore it.
    - If the first and last characters are letters, check whether they match; if so, remove both and test shorter string.
    - If last character isn't a letter, remove it and test shorter string.
    - If first character isn't a letter, remove it and test shorter string.

- To terminate the recursion, define a base case.
  - Strings with a single character are all palindromes.
  - The empty string is a palindrome.

Palindrome: Implement the recursion

- If possible, do not construct a new object on every method invocation.
  - Slow, wastes memory.
- Rather than constructing a smaller new string on each recursive method invocation, you better perform a tweak.
  - Use a global string (global class variable).
  - Use indices to specify the substring under test.
- To hide the tweak from the use, implement a (private) helper method

```
private boolean isPalindrome(int start, int end)
```

- Then public method for the user then gets

```
public boolean isPalindrome(){
  return isPalindrome(0, text.length() - 1);
}
```

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

BTI7055

Object-Oriented Programming 2

Generics

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS 2018

▶ Informatik

2nd part of the OOP2 Module

Topics

- Java Generics (Weeks 8—10)
- Java Collections Framework (Weeks 10—11)
- Parallel Programming with Threads (Weeks 12—14)
- Java Streams API (≠ Java I/O Streams) (Weeks 15—16)

- No class:
  - 11.5.2018 (Friday after Ascention Day/Auffahrt)
  - 16.5.2018 (BFH Sporttag – "The Games")

- Colloquium, Exam Preparation, Q&A: Last Friday 15.6.2018

Goals of 2nd Part OOP2

- You learn how to write
  - more robust code. → Generics
  - code more efficiently by reusing existing data structures. → Collections
  - exploit all processor cores of your system. → Threads
  - code at a higher level of abstraction in a functional style. → Streams API
    - Declarative Style: program describes only the "*what*" but not the "*how*".
    - Streams can be processed in parallel (transparently without the programmer having to explicitly use threads).

Topic 7: Generic Programming

Generic Programming in Java

Book Chapters:

- Chapter 18: "Generic Classes" in Cay S. Horstmann, "Big Java Early Objects", 5th Edition.
- Chapter 8: "Generic Programming" in Cay S. Horstmann, "Core Java", 10th Edition.
- "Item 31: Use Bounded Wildcards to Increase API Flexibility" in Joshua Bloch, "Effective Java", 3rd Edition.

Outline Topic 7: Generic Programming

- Generic Classes, Interfaces and Methods
- Raw Type and Type Erasure
- Type Bounds and Wildcards
- Variances: Covariant and Contravariant Types

Generic Programming

- Generic Programming allows implementing
  - Algorithms and
  - Data structures

  } Libraries

  independent of concrete data types.

- **Advantages:**
  - Improves code reuse.
  - Type checks at runtime (by voiding the use of down-casts) → increased type safety → fewer runtime exceptions.
  - Improves code readability.

Generic Programming and Java

- Generic Programming in Java means Generic Types (Parameterized Types). Types can have parameters, such called Type Parameters.
- Java Generic Types look similar to C++ class templates on the surface. However, they are based on different mechanisms and should not be confused with C++ templates.

- Java supports generic types only since Java 5.0 (2004).
- Generic types had to be added without breaking backward compatibility of the language!
  - Even today, you can mix code that uses generic code (code that uses type parameters) and non-generic code (code using raw type).
  - Adding support was non-trivial and has implications.
  - The Java Virtual Machine is unaware of Generic Types.

Users of Generic Types

- Generic Types are difficult! It is not easy to implement a generic class.
- Their introduction without breaking old code made Java programming more difficult.
  → Watch out for the corner cases that we will discuss today!

- Generic Programming at three skill levels:
  - Using existing generic classes (primarily from the Java Collections library).
  - Understanding error messages when mixing different generic classes.
  - Implementing your own generic classes and methods.

- Application programmers typically do not write a lots of generic classes themselves.
- They primarily make use of the Generics from Java Collections.

- We expect that you are 100% comfortable using Generic Classes (particularly, from the Collections library, discussed in the next topic)
- We expect that you are able to fix compiler errors related to Generics.
- You should be able to write your own generic classes using the provided guidelines.

Motivation: An ArrayList

A better Array List that knows how to grow as it would be implemented before Java 5.0 (without Generics):

```
public class ArrayList {
  private Object[] objects;
  private int size;

  public ArrayList() {
    objects = new Object[10];
    size = 0;
  }

  public void add(Object o) {
    if (objects.length == size) objects = java.util.Arrays.copyOf(objects, 2*size);
    objects[size++] = o;
  }

  public Object get(int index) {
    if (index < 0 || index >= size) throw new IndexOutOfBoundsException();
    return objects[index];
  }

  public void set(int index, Object o) {
    if (index < 0 || index > size) throw new IndexOutOfBoundsException();
    if (index == size) add(o);
    else objects[index] = o;
  }

  public int getSize() { return size; }
}
```

Problems with ArrayList

Type Safety: References are stored in array of type Object.

- A down-cast is needed for every get().
- One can store references to objects of any type.

- **Example:**

```
ArrayList names = new ArrayList();
names.add("Bob");
names.add("Joe");

String first = names.get(0);
String first = (String)names.get(0);

names.add(42);  // equivalent to names.add(Integer(42))
names.add(new JButton("click me"));
String second = (String)names.get(2);
```

Compile Time Error:
incompatible types: java.lang.Object
cannot be converted to java.lang.String

down-cast is needed

Objects of any type can be stored.

Runtime Error:
java.lang.ClassCastException: java.lang.Integer
cannot be cast to java.lang.String

→ **Generics offer a better solution: Type Parameters.**

Generic Class `ArrayList`

- Since Java 5.0 you can implement **`ArrayList`** a Generic Class.
- The generic class has a Type Parameter to specifies the element type of **`ArrayList`**.
- The list object can then be instantiated as follows:

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

← generic class `java.util.ArrayList`

- The compiler then knows that this list contains Strings only.
  - No down-casts necessary
  - The attempt to store a non-String object into the lists results in a compile-time error
    → Type safety is guaranteed.
  - **Example**:

```
names.add("Joe");
String first = names.get(0);

names.add(42);
names.add(new JButton("click me"));
```

← No cast necessary.

Compiler Error:

```
error: no suitable method found for add(int)
error: no suitable method found for add(javax.swing.JButton)
```

Syntactic Simplifications since Java SE 7

- Java has always been an extremely verbose programming language.

```
ArrayList<String> names = new ArrayList<String>()
```

- Since Java SE 7 you can safe some typing using the diamond operator.
  - The type parameter can be be omitted in the constructor.
  - The omitted type is inferred from the type of the variable.

Type omitted in constructor

```
ArrayList<String> names = new ArrayList<>()
```

- It's a start but type inference in Java is still very limited.
  Compare the same statement in Scala:

```
var names = new ArrayList[String]
```

Type Parameters

- Generic classes and Generic interfaces have one or more Type Parameters

Generic Class

Type Parameters K (for key), V (for value)

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
  ...
  public HashMap() { ... }
  public V put(K key, V value) { ... }
  public V get(Object key) { ... }
  ...
}
```

Generic Interface

Type Parameter E (for entry)

```
public interface List<E> extends Collection<E> {
  ...
  public E get(int index);
  public void add(E element);
  ...
}
```

Naming Conventions for Type Parameters

- Type Parameter can be any valid identifier, by convention, single upper-case letters are used:
  - `T` for a general generic type (for everything other than below)
  - `N` for numbers
  - `E` for elements in a collection (e.g., in a list, in a set, etc.)
  - `K` for keys in a map
  - `V` for values in a map
  - `S`, `U`, `W` for additional generic types

Putting Objects into Boxes

This is a is an illustration of the most simplistic collection, a box with a single object inside.

```
public class StringBox {
  private String content;

  public StringBox(String content) {
    this.content = content;
  }

  public void setContent(String content) {
    this.content = content;
  }

  public String getContent() {
    return content;
  }
}
...

public static void main(String args[]) {
  StringBox stringBox = new StringBox("Foo");
  String str = stringBox.getContent();
}
```

```
public class IntegerBox {
  private Integer content;

  public IntegerBox(Integer content) {
    this.content = content;
  }

  public void setContent(Integer content) {
    this.content = content;
  }

  public Integer getContent() {
    return content;
  }
}
...

public static void main(String args[]) {
  IntegerBox integerBox = new IntegerBox(42);
  int i = integerBox.getContent();
}
```

For reasons that will become clear later, we use `Integer` rather than the primitive `int` type.

A Generic Box

Box is a generic class with one type parameter:

```
public class Box<T> {
  private T content;

  public Box(T content) {
    this.content = content;
  }

  public void setContent(T content) {
    this.content = content;
  }

  public T getContent() {
    return content;
  }
}
```

T = String

T = Integer

T = Box<Integer>

Box objects storing storing objects of different types can be created out of the same generic class.

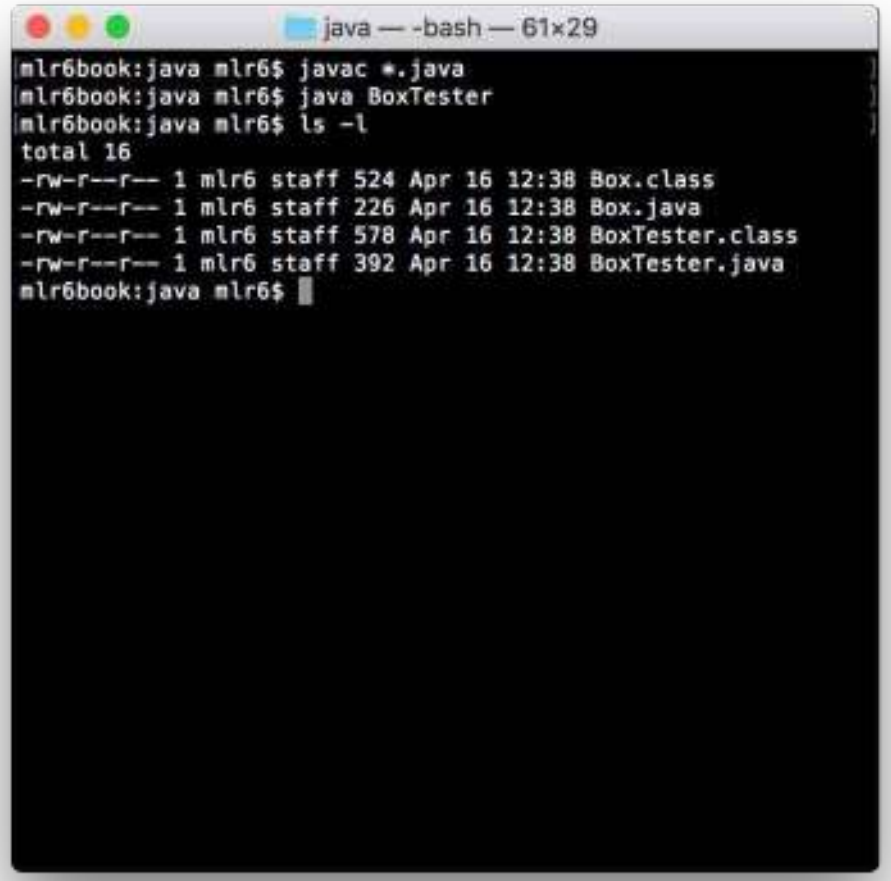
```
class BoxTester {

  public static void main(String[] args) {
    Box<String> stringBox = new Box<>("Foo");
    String str = stringBox.getContent();

    Box<Integer> integerBox = new Box<>(42);
    int i = integerBox.getContent();

    Box<Box<Integer>> boxWithIntegerInBox =
      new Box<>(integerBox);
    int j = boxWithIntegerInBox.getContent()
                                .getContent();
  }
}
```

Instantiating a Generic Type

- A Generic Type is instantiated by substituting the type parameters with concrete types.
- **Example**
  - **Box<String>**
  - Box<Integer>
  - Box<Shoe>

- Think of the resulting class (type) as having the constructor and methods:
  - `public Box(String content)`
  - `public void setContent(String content)`
  - `public String getContent()`

- However, there is still only one `.class` file, that for the generic class Box.



only one class file (= class) for Box

A Generic Class with two Type Parameters

```java
public class Pair<S, T> {
  private S first;
  private T second;

  public Pair(S first, T second) {
    this.first = first;
    this.second = second;
  }

  public S getFirst() {
    return first;
  }

  public T getSecond() {
    return second;
  }

  public void setFirst(S first) {
    this.first = first;
  }

  public void setSecond(T second) {
    this.second = second;
  }
}
```

```java
public class PairTester {
  public static void main(String[] args) {
    Pair<String, Integer> pair1 =
                        new Pair<>("Foo", 42);
    String str1 = pair1.getFirst();
    int i1 = pair1.getSecond();

    Pair<Integer, String> pair2 =
                        new Pair<>(1, "Bar");
    int i2 = pair2.getFirst();
    String str2 = pair2.getSecond();
  }
}
```

Type Parameters

- Type Parameters in Generic Classes can be Interfaces and Classes
- **Examples:**
  - `Box<String>` — `String` is a class
  - `Box<EventListener>` — `EventListener` is an Interface
  - `Box<Box<Integer>>` — `Box<Integer>` is a generic class with a concrete type `Integer`

- **Note**: primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `double`, `float`) **cannot** be used as type parameters!
  - This is because the Java Types System differentiates between primitive and object types. Java's Type System.
  - Use the wrapped object types instead: `Boolean`, `Byte`, `Char`, `Short`, `Integer`, `Long`, `Double`, `Float`
  - ~~`Box<int>`~~ → `Box<Integer>`

Question 1: java.util.HashMap

- The Java standard library provides a class `HashMap<K,V>` that maps entries with key type `K` to values type `V`.

- Instantiate a hash map that maps strings to an array of integers.
- Store a reference to this object in a variable `map`.

Question 2

- What is the difference between

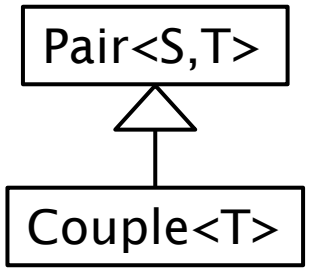
```
ArrayList<Pair<String, Integer>>
```

and

```
Pair<ArrayList<String>, Integer>
```

?

Inheriting Generic Classes



Couple is a special Pair where first and second are both of the same type.

```
class Couple<T> extends Pair<T, T> {
  public Couple(T first, T second) {
    super(first, second);
  }

  public void swap() {
    T t = getFirst();
    setFirst(getSecond());
    setSecond(t);
  }

  public List<T> toList() {
    List<T> list = new ArrayList<>();
    list.add(getFirst());
    list.add(getSecond());
    return list;
  }
}
```

Usage of Couple Class

```
public class static CoupleTester {
  public static void main(String[] args) {
    Couple<Integer> couple = new Couple<>(42, 7);
    int i1 = couple.getFirst();   // => 42
    int i2 = couple.getSecond();  // => 7
    couple.swap();
    List<Integer> list = couple.toList();
                                  // => [7, 4]
  }
}
```

Generic Methods

- Generic Methods specify their own type parameters.
  - Scope of type parameter is limited to method.
  - Generic methods can be defined in generic and ordinary (non-generic) classes.
  - Static and non-static generic methods are allowed.
  - Generic constructors are also allowed.

- **Syntax**: the type variables are defined after public, static, etc. but before the return type.

```
public static <T> T someMethod (T t) { ... } // method with generic
                                             // argument and return value
```

- **Application**: Ensure type safety between argument and return types.

Example: Array Utility Class with Static Generic Methods

```
public class ArrayUtil {

  public static <E> void printAll(E[] array) {
    StringJoiner joiner = new StringJoiner(", ");
    for(E e : array) {
      joiner.add(e.toString());
    }
    System.out.println(joiner.toString());
  }

  public static <E> E getMiddle(E[] array) {
    return array[array.length/2];
  }
} ...
  String[] names = {"Alice", "Bob", "Charlie"};
  Rectangle[] rects = {new Rectangle(10, 20, 30, 40)};

  String middleStr = ArrayUtil.<String>getMiddle(names);

  Rectangle middleRect = ArrayUtil.getMiddle(rects);

  ArrayUtil.printAll(names);
  ArrayUtil.printAll(rects);
```

A non-generic, ordinary class

Generic method with type parameter E (before return type)

Generic method with type parameter E (before return type)

type witness

The type parameter is specified during the invocation of the method.

In most cases, the compiler can infer the value of the parameter type from the arguments. The type witness can then be omitted. Since the type of `rect` is `Rectangle[]`, the compiler infers E=`Rectangle`.

Bounded Type Parameters

Add min() Method to ArrayUtil

```
public class ArrayUtil {

  public static <E> E min(E[] array) {
    if (array.length == 0) {
      throw new IllegalArgumentException("empty array");
    }
    E smallest = array[0];
    for (int i = 0; i < array.length; i++) {
      if (array[i].compareTo(smallest) < 0) {
        smallest = array[i];
      }
    }
    return smallest;
  }
}
```

Compiler Error

```
Error:(42, 19) java: cannot find symbol
  symbol:   method compareTo(E)
  location: class java.lang.Object
```

Problem: Compiler cannot guarantee that type E has a `compareTo()` method. Compiler needs to be conservative and assume E is most basic type (`Object`), but object does not have `compareTo()` method.

Reason: E is unbounded. It can be any type.
→ We need to restrict the type parameter so that it can only take values to those classes that have a `compareTo()` method.

Min() with Bounded Type Parameter

```
public class ArrayUtil {

  public static <E extends Comparable> E min(E[] array) {
    if (array.length == 0) {
      throw new IllegalArgumentException("empty array");
    }
    E smallest = array[0];
    for (int i = 0; i < array.length; i++) {
      if (array[i].compareTo(smallest) < 0) {
        smallest = array[i];
      }
    }
    return smallest;
  }
}
```

E is bounded.
Type used for E must extend Comparable.

Note: the extends keyword is used here also for interfaces. (Comparable is an interface).
→ E must implement the Comparable interface.

Min() with Bounded Type Parameter: Compiler Warning

→ use of Generics Skill-Level 2: understanding error messages

```
$ javac ArrayUtil.java
Note: ArrayUtil.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

$ javac -Xlint:unchecked ArrayUtil.java
ArrayUtil.java:42: warning: [unchecked] unchecked call to compareTo(T) as a member
of the raw type Comparable
      if (array[i].compareTo(smallest) < 0) {
                           ^
  where T is a type-variable:
    T extends Object declared in interface Comparable
1 warning
```

Reason: Comparable Interface is Generic

```java
public interface Comparable<T> {
  /**
   * Compares this object with the specified object for order.  Returns a
   * negative integer, zero, or a positive integer as this object is less
   * than, equal to, or greater than the specified object.
   *
   * @param   o the object to be compared.
   * @return  a negative integer, zero, or a positive integer as this object
   *          is less than, equal to, or greater than the specified object.
   */
   public int compareTo(T o);
}
```

Presumably corrected min() with Bounded Type Parameter

Specify E as the type for Comparable.

```
public class ArrayUtil {

  public static <E extends Comparable<E>> E min(E[] array) {
    if (array.length == 0) {
      throw new IllegalArgumentException("empty array");
    }
    E smallest = array[0];
    for (int i = 0; i < array.length; i++) {
      if (array[i].compareTo(smallest) < 0) {
        smallest = array[i];
      }
    }
    return smallest;
  }
}
```

→ The warning disappears but this is still not 100% correct.
(correct: `public static <E extends Comparable<? super E>> E min(E[] array)`)

We need to learn more about type bounds, variances, and wildcards.

Bounded Type Parameters

- In `class Box<T> { }`, the type parameter T is unbound.
  - There are no restrictions on the type used in instantiations.

- The declaration `class Box<T> { }`, is a shortcut for `class Box<T extends Object> { }`.
  - Only methods from class `Object` can be called on objects of type T.

- A Bounded Type Parameter T must be subtype of some bounding type.
- **Syntax:** `<T extends BoundingType>`
  - `BoundingType` can be a class or interface, extends means "`extends` or `implements`".

Example: Generic vs. Non-Generic Collections (1)

```java
public interface Measurable {
  double getMeasure();
}
class Length implements Measurable { ... }
class Weight implements Measurable { ... }

class GenericMeasurables<T extends Measurable> {

  private List<T> items = new ArrayList<>();

  public void addItem(T item) {
    items.add(item);
  }

  public double getTotal() {
    double sum = 0;
    for (T item : items) {
      sum += item.getMeasure();
    }
    return sum;
  }
}
```

```java
class NonGenericMeasurables {

  private List<Measurable> items = new ArrayList<>();

  public void addItem(Measurable item) {
    items.add(item);
  }

  public double getTotal() {
    double sum = 0;
    for (Measurable item : items) {
      sum += item.getMeasure();
    }
    return sum;
  }
}
```

Example: Generic vs. Non-Generic Collections (2)

```
class Length implements Measurable { ... }
class Weight implements Measurable { ... }

public class MeasurableTest {
  public static void main(String[] args) {
    GenericMeasurables<String> genericStrings = new GenericMeasurables<>();
    GenericMeasurables<Length> genericLengths = new GenericMeasurables<>();
    GenericMeasurables<Weight> genericWeights = new GenericMeasurables<>();
    NonGenericMeasurables nonGenericMeasurables = new NonGenericMeasurables();

    Length length = new Length(1.25);
    Weight weight = new Weight(23.0);

    genericLengths.addItem(length); // OK
    genericLengths.addItem(weight); // weight cannot be added to a GenericMeasureables<Length>
    genericWeights.addItem(weight); // OK
    genericWeights.addItem(length); // length cannot be added to a GenericMeasureables<Weight>
    nonGenericMeasurables.addItem(length);  // OK: Length is a Measurable
    nonGenericMeasurables.addItem(weight);  // OK: Weight is a Measurable
  }
}
```

Error: type argument String is not within bounds of type-variable T

← can only store Lengths

← can only store Weights

can store any Measurable

Bounded Type Parameters with Multiple Bounds

- Two or more type bounds can be specified.
  Example: `<T extends Serializable & Cloneable>`
  - `T` must be serializable and cloneable

- If multiple bounds are specified and one of them is a class, the class must be specified first:

```
public class A { }
public interface B { }
public interface C { }

public class D<T extends A & B & C> { }  // OK, class A at first place
public class E<T extends B & A & C> { }  // Not OK, class A not at first place
```

Generic Code in the Virtual Machine

Type Erasure, Backward-compatibility, Raw Types

Type Erasure

- The concrete values of the type parameters are erased during compilation.
  - Result of the backward compatibility requirement when Generics were introduced in Java 5.
  - The VM does not need to know about Generics.
    This will most likely change in the future JDK >10 (see Project Valhalla, with support for Value Types and and Generic Specialization).
  - Type erasure also applies to other languages that are based on the JVM, e.g., Scala.

- The type parameters are replaced by their upper bound type.
  - `<T>` becomes `Object` since `<T>` corresponds to `<T extends Object>`
  - `<T extends Number>` becomes `Number`
  - `<T extends Measurable & Serializable>` becomes `Measurable`

- How can generics be implemented in a type-safe manner if the VM is unaware of them?
  - Compiler inserts corresponding type casts into the bytecode.
  - Compiler inserts bridge methods to guarantee polymorphism (e.g., overriding of methods) works correctly.

Example Type Erasure: Box<T>

```java
public class Box<T> {
  private T content;

  public Box(T content) {
    this.content = content;
  }

  public void setContent(T content) {
    this.content = content;
  }

  public T getContent() {
    return content;
  }
}

public class BoxTester {
  public static void main(String[] args) {
    Box<String> stringBox = new Box<>("Foo");
    String s = stringBox.getContent();
  }
}
```

Decompiled Bytecode

Corresponding decompiled Bytecode from Box.class and BoxTester.class.:

```java
public class Box {
  private Object content;
  public Box(Object content) {
    this.content = content;
  }
  public void setContent(Object content) {
    this.content = content;
  }
  public Object getContent() {
    return content;
  }
}

public class BoxTester {
  public static void main(String[] args) {

    Box stringBox = new Box("Foo");
```

No cast needed here.

```java
    String s =  (String)stringBox.getContent();
  }
}
```

Compiler introduces cast.

Example Type Erasure: Box<T extends Number>

```java
public class Box<T extends Number> {
  private T content;

  public Box(T content) {
    this.content = content;
  }

  public void setContent(T content) {
    this.content = content;
  }

  public T getContent() {
    return content;
  }
}

public class BoxTester {
  public static void main(String[] args) {
    Box<String> stringBox = new Box<>("Foo");
    String s = stringBox.getContent();
  }
}
```

Decompiled Bytecode

Corresponding decompiled Bytecode from Box.class and BoxTester.class.:

Resulting type is upper bound (most general type from type bound)

```java
public class Box {
  private Number content;
  public Box(Number content) {
    this.content = content;
  }
  public void setContent(Number content) {
    this.content = content;
  }
  public Number getContent() {
    return content;
  }
}

public class BoxTester {
  public static void main(String[] args) {

    Box intBox = new Box(Integer.valueOf(42));
    Integer i =  (Integer)intBox.getContent();
  }
}
```

No cast needed here.

Compiler introduces cast.

Raw Type: Generic classes can be used like Ordinary Classes

- A raw type is the name of a generic class or interface with any type arguments.
  Example: **Box is the raw type of the generic class Box<T>**
- Raw types are used to express generic types for pre Java 5 code (backward compatibility)
- **Note**: Raw type **Box** and and type **Box<Object>** are not the same for the compiler.

no type values specified → rawBox has raw type of Box<T>, Box

```
public class RawBoxTester {
  public static void main(String[] args) {
    Box rawBox = new Box("Foo");                  // reference has type Box
    String s = (String)rawBox.getContent();       // OK   (rawBox has upper bound Object)
    rawBox.setContent(42);                        // OK   (Integer is an object)
    Integer i = (Integer)rawBox.getContent();     // OK

    Box<String> stringBox = new Box<>("Foo");     // reference has type Box<String>
    rawBox = stringBox;                           // OK: a Box is a Box<String>
    rawBox.setContent(7);                         // OK!
    Box<Object> objectBox = new Box<>("Bar");     // OK: "Bar" is an Object
    objectBox.setContent(1);                      // OK: Integer is an Object
  }
}
```

rawBox has static type Box (raw type) and dynamic type Box<Object> (after erasure)

Do not use Raw Types

- Raw types are only used today to interface with old pre Java 5 code.
- The use of raw types disables the type system that provides additional type safety.
  - This is no worse than the situation in pre-Java 5 era though.

- The compiler warns you about this.

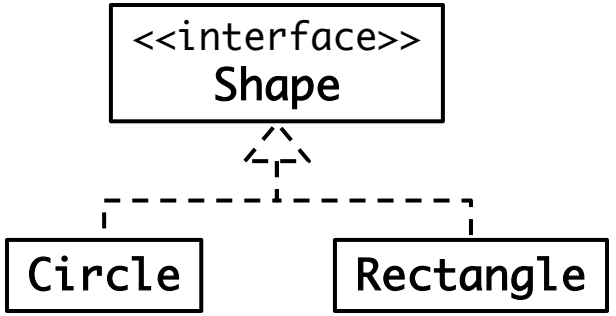
```
$ javac RawBoxTester.java
Note: RawBoxTester.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

- Get more detailed information using compiler flag:

```
RawBoxTester.java:3: warning: [unchecked] unchecked call to Box(T) as a member of the raw type Box
    Box rawBox = new Box("Foo");
                 ^
  where T is a type-variable:
    T extends Object declared in class Box
RawBoxTester.java:5: warning: [unchecked] unchecked call to setContent(T) as a member of the raw type Box
    rawBox.setContent(42);                        // OK
                      ^
  where T is a type-variable:
    T extends Object declared in class Box
```

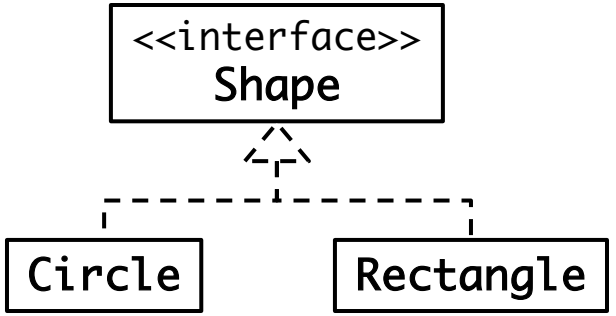
Wildcard Types

Example: A Drawing Application (1)



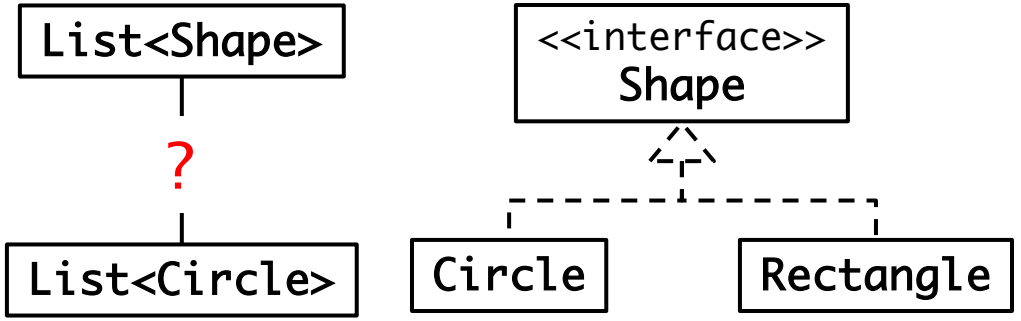

```
public interface Shape {
  public abstract void draw(Canvas canvas);
}

public class Circle implements Shape {
  private int x, y, radius;
  public Circle(int x, int y, int radius) {
    this.x = x; this.y = y; this.radius = radius;
  }
  @Override
  public void draw(Canvas canvas) { System.out.println("draw Circle"); }
}

public class Rectangle implements Shape {
  private int leftX, topY, width, height;
  public Rectangle(int leftX, int topY, int width, int height) {
    this.leftX = leftX; this.topY = topY; this.width = width; this.height = height;
  }
  @Override
  void draw(Canvas canvas) { System.out.println("draw Rectangle"); }
}
```

Example: A Drawing Application (2)




```
public class Canvas {
  public void draw(Shape shape) {
    shape.draw(this);
  }

  public void drawAll(List<Shape> shapes) {
    for (Shape shape : shapes) {
      shape.draw(this);
    }
  }

  public static void main(String[] args) {
    Canvas canvas = new Canvas();

    List<Shape> shapes = Arrays.asList(new Circle(0, 0, 100), new Rectangle(-10, 10, 20, 20));
    canvas.drawAll(shapes);




  }
}
```

Example: A Drawing Application (3)




```
public class Canvas {
  public void draw(Shape shape) {
    shape.draw(this);
  }

  public void drawAll(List<Shape> shapes) {
    for (Shape shape : shapes) {
      shape.draw(this);
    }
  }

  public static void main(String[] args) {
    Canvas canvas = new Canvas();

    List<Shape> shapes = Arrays.asList(new Circle(0, 0, 100), new Rectangle(-10, 10, 20, 20));
    canvas.drawAll(shapes);

    List<Circle> circles = Arrays.asList(new Circle(0, 0,  10), new Circle(10, 10, 20));
    canvas.drawAll(circles);
  }
}
```

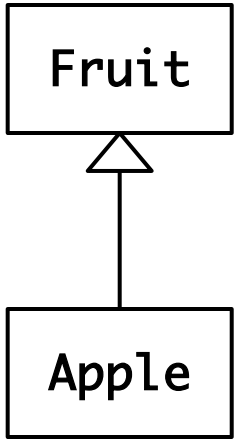
```
Error:(49, 20) java: incompatible types:
java.util.List<ch.bfh.ti.oop2.generics.Circle> cannot be converted
to java.util.List<ch.bfh.ti.oop2.generics.Shape>
```

Wait, what ...?
A List<Circle> is not a List<Shape>?
What is going on here?

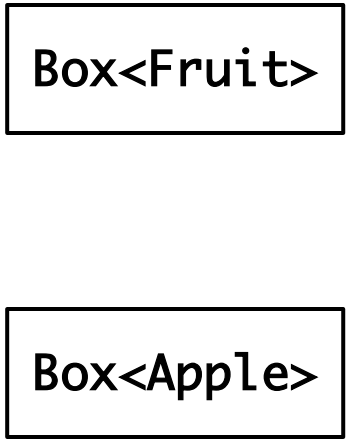
Variances

```
public abstract class Fruit { }
public class Apple extends Fruit { }
public class Box<T> {
  T content;
  Box(T content) { this.content = content; }
}
```

An Apple is a Fruit.

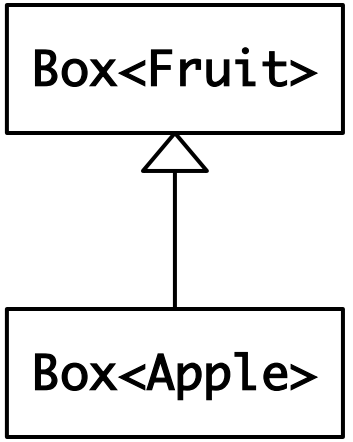


A Box<Apple> is un-related to Box<Fruit>.



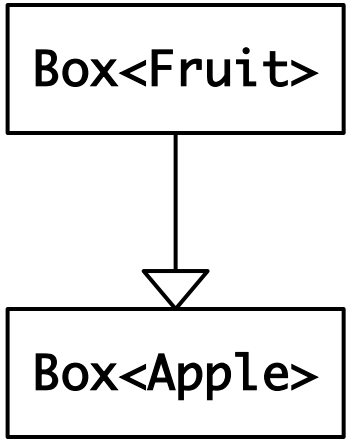
Invariant

A Box<Apple> is a Box<Fruit>.



Covariant

A Box<Fruit> is a Box<Apple>.



Contravariant

Variances in Java

- Variances
  - Declaration-site Variance: Variance determined at the declaration site of the type parameter (→ where you defined the **Box** class). It is a property of the class or the interface.
  - Use-site Variance: Variance determined at the site where the type parameter is used (→ where you defined a reference of `Box<Apple>`). It is a property of the reference.

- The concept of variances supported is fairly primitive in Java:
  - Java supports only use-site variance (Scala supports both).
  - Generic classes and interfaces are **invariant** → Hence, the error when calling `drawAll(..)`

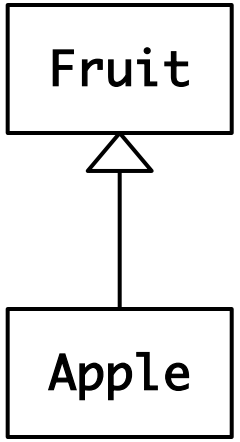
- Plain arrays are covariant in Java, i.e., `Apple[]` is a `Fruit[]`.
  - This is a bug in the language.
  - This is not type-safe as it can lead to an `ArrayStoreException` (see OOP1 course).
  - Arrays are also C# (what a surprise!) and Swift. ☹
  - Arrays are invariant in Kotlin and Scala. ☺

Variances

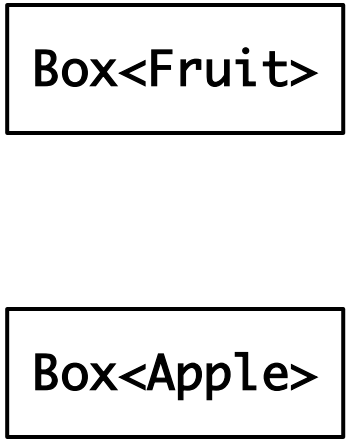
```
public abstract class Fruit { }
public class Apple extends Fruit { }
public class Box<T> {
  T content;
  Box(T content) { this.content = content; }
}
```

Generic classes and interfaces are invariant in Java.

An Apple is a Fruit.

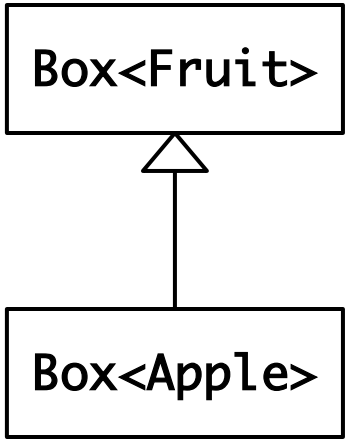


A Box<Apple> is un-related to Box<Fruit>.



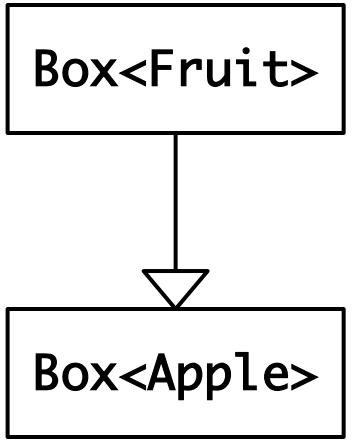
Invariant

A Box<Apple> is a Box<Fruit>.



Covariant

A Box<Fruit> is a Box<Apple>.



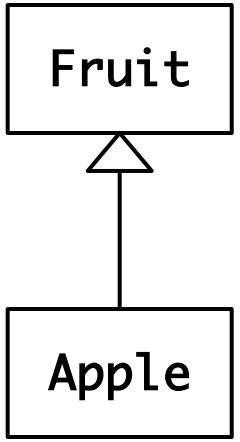
Contravariant

Wildcard Type to Express Use-Site Variance

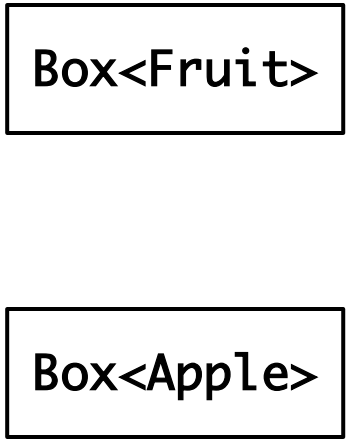
Use-site Variances can be expressed with bounds on the **Wildcard Type** ?.

```
public abstract class Fruit { }
public class Apple extends Fruit { }
public class Box<T> { ... }
...
Box<Fruit> boxFruit = ...
Box<Apple> boxApple = ...
```

An `Apple` is a `Fruit`.

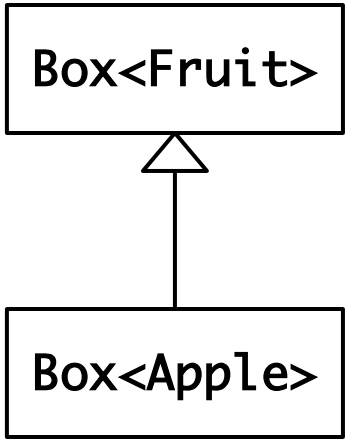


A `Box<Apple>` reference is unrelated to `Box<Fruit>` reference.



Invariant

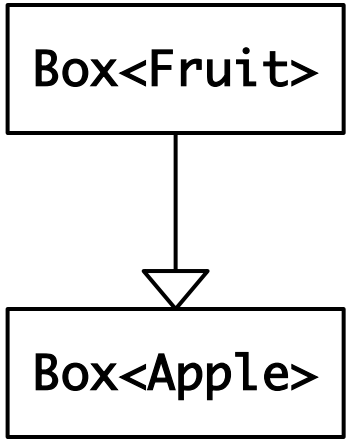
A `Box<Fruit>` reference can is point to a `Box<Apple>` object.



Covariant

`<? extends Fruit>`

A `Box<Apple>` reference can is point to a `Box<Fruit>` object.



Contravariant

`<? super Apple>`

Wildcard Types (1)

- The wildcard **?** represents an unknown type, for example **Box<?>**.
- Wildcard types can be bounded:
  - Upper Bound: ? extends SomeType → **?** cannot stand for any type higher up in the type hierarchy. **?** must at least be a SomeType.
  - Lower Bound: ? super SomeType → **?** cannot stand for any type lower (a more specific) in the type hierarchy. **?** may be at most a SomeType.

Name	Syntax	Meaning
Wildcard with upper bound	? extends SomeType	Any subtype of SomeType or SomeType itself
Wildcard with lower bound	? super SomeType	Any supertype of SomeType or SomeType itself
Unbounded wildcard	?	Any type

- Note that Box<?> (unknown type) and Box (raw type) are not the same, because Box<?> guarantees type safety of some unknown type.
- Box<?> is a shortcut for Box<? extends Objects>.

Wildcard Types (2)

- Wildcards **?** cannot be used as normal type, i.e., you cannot define a variable "`? a`".
- Wildcards **?** can be used as values for type parameters, i.e., "`List<?> a`".
- Wildcards can be used in declaration of ...
  - local variables
  - instance variables
  - static variables
  - method parameters, e.g., "`List<?> a`"
  - return values

- Using wildcards in return types is not recommended because it forces programmers using the code to deal with wildcards, e.g., List<?> is the return type.
- Wildcards are often used to bypass the restrictions of the strict subtype rules (while avoiding raw types).
- Note that wildcards cannot be used in the definition of generic classes or when invoking their constructors, or a generic methods.

Modified Drawing Application

```
public class Canvas {
  public void draw(Shape shape) {
    shape.draw(this);
  }

  public void drawAll(List<? extends Shape> shapes) {
    for (Shape shape : shapes) {
      shape.draw(this);
    }
  }

  public static void main(String[] args) {
    Canvas canvas = new Canvas();

    List<Shape> shapes = Arrays.asList(new Circle(0, 0, 100), new Rectangle(-10, 10, 20, 20));
    canvas.drawAll(shapes);

    List<Circle> circles = Arrays.asList(new Circle(0, 0,  10), new Circle(10, 10, 20));
    canvas.drawAll(circles);     // No error!

    List<Rectangle> rectangles = Arrays.asList(new Rectangle(-10, 10, 20, 20));
    canvas.drawAll(rectangles);   // No error!
  }
}
```

shapes is a List of Shape or subclass of Shape objects.

Quiz: Bounded Wildcards – Find all Errors and Warnings!

```
Apple apple = new Apple();
Banana banana = new Banana();
Fruit fruit = apple;

Box<Fruit> fruitBox = new Box<>(fruit);
Box<Apple> appleBox = new Box<>(apple);
Box<Banana> bananaBox = new Box<>(banana);

fruitBox = appleBox;
appleBox = fruitBox;
appleBox = bananaBox;

Box rawBox = new Box(apple);
rawBox = fruitBox;
rawBox = appleBox;

Box<? extends Fruit> covariantFruitBox = new Box<>(fruit);
covariantFruitBox = appleBox;
covariantFruitBox = bananaBox;
covariantFruitBox = fruitBox;
covariantFruitBox = rawBox;
rawBox = covariantFruitBox;

Box<? super Apple> contravariantFruitBox = new Box<>(apple);
contravariantFruitBox = fruitBox;
contravariantFruitBox = appleBox;
contravariantFruitBox = bananaBox;
contravariantFruitBox = rawBox;
rawBox = contravariantFruitBox;
```

A Fruit Basket – Initial Attempt

```
public class FruitBasket{
  private List<Fruit> fruits = new ArrayList<>();

  public FruitBasket() { }

  /** Add single fruit to basket. */
  public void addFruit(Fruit fruit) {
    fruits.add(fruit);
  }

  /** Add multiple fruits to basket. */
  public void addFruits(Collection<Fruit> moreFruits) {
    fruits.addAll(moreFruits);
  }

  /** Extract fruits in basket into basketContent collection. */
  public void extractFruits(Collection<Fruit> basketContent) {
    for (Fruit fruit : fruits) {
      basketContent.add(fruit);
    }
  }
}
```

Fruit Basket User (1)

```
List<Object> objects = new ArrayList<>();
List<Fruit> fruits = new ArrayList<>();
List<Apple> apples = Arrays.asList(new Apple(), new Apple(), new Apple());

FruitBasket fruitBasket = new FruitBasket();

fruitBasket.addFruit(new Apple()); // OK
fruitBasket.addFruits(fruits);     // OK
fruitBasket.addFruits(apples); // Error: List<Apples> is not a Collection<Fruit>

apples.clear();

fruitBasket.extractFruits(apples);  // Error: List<Apples> is not a Collection<Fruit>
fruitBasket.extractFruits(fruits);  // OK
fruitBasket.extractFruits(objects); // Error: List<Object> is not a Collection<Fruit>
```

Which errors are justified?
Which errors should be fixed?

Fruit Basket User (2)

```
List<Object> objects = new ArrayList<>();
List<Fruit> fruits = new ArrayList<>();
List<Apple> apples = Arrays.asList(new Apple(), new Apple(), new Apple());

FruitBasket fruitBasket = new FruitBasket();

fruitBasket.addFruit(new Apple()); // OK
fruitBasket.addFruits(fruits);     // OK
fruitBasket.addFruits(apples); // Error: List<Apples> is not a Collection<Fruit>

apples.clear();

fruitBasket.extractFruits(apples);  // Error: List<Apples> is not a Collection<Fruit>
fruitBasket.extractFruits(fruits);  // OK
fruitBasket.extractFruits(objects); // Error: List<Object> is not a Collection<Fruit>
```

A list of List<Apples> is a collection that contains Apples which are Fruits. No problem → fixable error.

A list of List<Objects> is a collection that can store Objects, any Object, therefore Fruits too. No problem → fixable error.

Fruit Basket User (3)

```
List<Object> objects = new ArrayList<>();
List<Fruit> fruits = new ArrayList<>();
List<Apple> apples = Arrays.asList(new Apple(), new Apple(), new Apple());

FruitBasket fruitBasket = new FruitBasket();

fruitBasket.addFruit(new Apple()); // OK
fruitBasket.addFruits(fruits);     // OK
fruitBasket.addFruits(apples); // Error: List<Apples> is not a Collection<Fruit>

apples.clear();

fruitBasket.extractFruits(apples);  // Error: List<Apples> is not a Collection<Fruit>
fruitBasket.extractFruits(fruits);  // OK
fruitBasket.extractFruits(objects); // Error: List<Object> is not a Collection<Fruit>
```

What about this error? Is it fixable?

No! The fruit basket contains Fruits, any Fruit, not just Apples. It can also contain Bananas and Oranges.
We cannot add a Banana to a List<Apple>.
There is no guarantee that the basket contains only Apples.
→ Error not fixable, it is justified.

Fruit Basket – Insights

- `addFruits(Collection<Fruit> moreFruits)`
  - Must be able to accept `Collection<Apple>`, `List<Banana>`, etc.
  - Must not accept `Collection<Object>` or `List<Bacon>`

- `extractFruits(Collection<Fruit> basketContent)`
  - Must be able to accept `Collection<Fruit>`, `List<Object>`, etc.
  - Must not accept `Collection<Apple>` or `List<Banana>`

- What is the difference between these two?
  - `moreFruits` is an INPUT, i.e., `moreFruits` produces the elements for `FruitBasket`.
  - `basketContent` is an OUTPUT, i.e., `basketContent` consumes elements from `FruitBasket`.

Fixed FruitBasket

```
public class FruitBasket{
  private List<Fruit> fruits = new ArrayList<>();

  public FruitBasket() { }

  /** Add single fruit to basket. */
  public void addFruit(Fruit fruit) {
    fruits.add(fruit);
  }

  /** Add multiple fruits to basket. */
  public void addFruits(Collection<? extends Fruit> moreFruits) {
    fruits.addAll(moreFruits);
  }

  /** Extract fruits in basket into basketContent collection. */
  public void extractFruits(Collection<? super Fruit> basketContent) {
    for (Fruit fruit : fruits) {
      basketContent.add(fruit);
    }
  }
}
```

PECS Principle: Producer Extends, Consumer Super

Mnemonic based on Joshua Bloch, "Effective Java", Addison Wesley.

- For maximum flexibility, use wildcard types on input parameters that represent producers or consumers.

- Remember this simple rule:
  - PECS = Producer Extends, Consumer Super

- If a parameterized type represents a T producer, use <? extends T>.
  It is ok if it also provides more (more specific) than T.
- If a parameterized type represents a T consumer, use <? super T>.
  It is ok if it also accepts less (less specific) than T.

Liskov Substitution Principle

- Remember: If the user of a class has to think about wildcard types, there is probably something wrong with its API.

Bounded Type Parameter vs. Upper Bound Wildcard

```
public class ListUtil {
  public static <T extends Number> void processList(List<T> someList) {
    double sum = 0;
    for (Number n : someList) {
      sum += n.doubleValue();
    }
    System.out.println(sum);

    T first = someList.get(0);       // OK: create local reference of type T
    someList.add(someList.get(0));   // OK: modify list
  }

  public static void processListWildcard(List<? extends Number> someList) {
    double sum = 0;
    for (Number n : someList) {
      sum += n.doubleValue();
    }
    System.out.println(sum);

    ? first = someList.get(0);         // ERROR: ? is not a valid type
    someList.add(someList.get(0));     // ERROR: cannot add element to list
  }
}
```

Limitations of Generic Programming

Implications of Type Erasure

Type Parameter Cannot be Primitive Type

- Type Parameters cannot be substituted by primitive values.
  (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`)

- After Type Erasure, the resulting type is `Object` (or the upper bound).
- `Object` is a reference type.
- Primitive types have special status in the JVM. → They require special bytecode instructions.

- Used the wrapper classes `Boolean`, `Byte`, `Char`, `Short`, `Integer`, `Long`, `Float`, and Double instead as Type Parameter values.

- Scala does not suffer from this problem as there are no primitive types that are treated specially. Essentially, an `Int` in Scala is an `Integer` in Java.

Type Parameters Cannot be Instantiated

- It is impossible to create instances of type parameters.
- After type erasure, `new T()` would correspond to `new Object()`.

```
public static <T> void fill(T[] array) {
  for (int i = 0; i < array.length; i++)
    array[i] = new T(); // Error: type parameter T cannot be instantiated.
}
```

- Workaround: Let caller provide a default value. Note all array elements will point to this same default value, so it better be immutable.

```
public static <T> void fillWithDefault(T[] array, T defaultValue) {
  for (int i = 0; i < array.length; i++)
    array[i] = defaultValue;
}
```

Better Workaround since Java 8

- Make caller provide a constructor expression.
- Use the functional interface **Supplier<T>** for a function with no arguments and a result of type T.

```
import java.util.function.Supplier;

@FunctionalInterface
public interface Supplier<T> {
    T get();
}

public static <T> void fill(T[] array, Supplier<T> constr) {
  for (int i = 0; i < array.length; i++)
    array[i] = constr.get();
}

public static void main(String[] args ) {
  String[] array = new String[10];
  fill(array, String::new);        // Provide constructor reference
}
```

No Arrays of Type Parameter

- It is impossible to create arrays of type parameters.
- JVM needs concrete type in order to instantiate an array.

```
public class ArrayList<E> implements List<E> {

  private E[] elements;  // OK
  private int size;

  public ArrayList() {
    elements = new E[10]; // Error: type parameter E cannot be instantiated directly.
    elements = (E[])new Object[10];   // OK, instead!
    size = 0;
  }

  @Override
  public E get(int index) {
    return elements[index];
  }
  ...
}
```

No Arrays of Generic Types

- You cannot instantiate an array of Generic Types either.

```
List<String>[] arrayOfStringLists = new ArrayList<String>[10];
```

- Error Message: "Generic Array Creation"

- Arrays remember the component type. This allows them to check store operations and throw an **ArrayStoreException** if the type of the element to be written is not a subtype of the arrays component type

- If arrays would be allowed:
  After Type Erasure `arrayOfStringLists` is a `List[]` (array of raw type)

```
Object[] objects = arrayOfLstringLists;
objects[0] = "Hello";          // Error -- Component type is List. Good, error detected.
objects[0] = new ArrayList<BankAccount>();  // Error would not be detected!
```

Runtime Type Inquiry Only Works with Raw Types

- Since the type parameters are erased they cannot be queried.

```
if (a instanceof Pair<String>) // Error
```

- Also:

```
if (a instanceof Pair<T>) // Error
```

- And the cast:

```
List<String> ls = (List<String>)l; // Warning - can only test that a is a List
```

Type Variables Cannot Be Referenced in Static Fields or Static Methods

```
public class Singleton<T> {

  private static T singleInstance; // Error

  public static T getSingleInstance() {// Error
    if (singleInstance == null) {
      // construct new instance
    }
    return singleInstance;
  }
}
```

- Otherwise, a `Singleton<Random>` could be shared with a `Singleton<DataCache>`.

Method Overloading

- A class cannot have two overloaded methods that will have the same signature after Type Erasure.

```
public class ListPrinter {
  public void print(List<String> strList) { }  // OK
  public void print(List<Integer> intList) { } // Name clash print(List<Integer>) and
                                               // print(List<String>) have same erasure
  public void print(List<Double> intList) { }  // Name clash print(List<Double>) and
                                               // print(List<String>) have same erasure
}
```

Summary: Generic Programming in Java

- Java Generics allows you to write code once and use it many times for different types of objects.
- Generic classes and Interfaces are the basics to build a powerful set of algorithms and collections.
- Generic Classes have one or more type parameters.
- Type parameters are instantiated with concrete type values.
- Type parameters can have upper bounds but no lower bounds.
- Type parameters are erased by the compiler, i.e., the actual value of a type parameter is not available at runtime. → Type Erasure
- Type erasure introduces several restrictions on how generic types can be used.

- An instance of `ListArray<String>` can have different types:
  - Type `ListArray<String>` at compile type, i.e., type parameter has value `String`
  - Wildcard type such as `List<?>`
  - The raw type `List`.
- Wildcard types can be bounded too, they can have upper and lower bounds.
- Remember the PECS principle for input and output parameters.

Appendix: C++ Templates

No Type Erasure in C++ for Template Classes

Generic Box in C++

```
template<typename T>
class Box
{
    T content;
  public:
    Box(const T &t): content(t) { }
    void setContent(const T &t) { content = t; }
    T getContent() const { return content; }
};

int main()
{
  auto intBox = Box<int>(42);       // instantiate a box that contains ints (primitive types)
  intBox.setContent(1);
  int i = intBox.getContent();

  auto floatBox = Box<float>(1.23); // instantiate a box that contains floats (primitive types)
  floatBox.setContent(3.14);
  float f = floatBox.getContent();

  return 0;
}
```

Generated Code from C++ Templates

```
$ clang++ -std=c++11 -c box.cc
$ nm -gU box.o
0000000000000120 T __ZN3BoxIfE10setContentERKf
00000000000000f0 T __ZN3BoxIfEC1ERKf
0000000000000180 T __ZN3BoxIfEC2ERKf
00000000000000c0 T __ZN3BoxIiE10setContentERKi
0000000000000090 T __ZN3BoxIiEC1ERKi
0000000000000160 T __ZN3BoxIiEC2ERKi
0000000000000140 T __ZNK3BoxIfE10getContentEv
00000000000000e0 T __ZNK3BoxIiE10getContentEv
0000000000000000 T _main
```

← Show defined symbol in object file.

Separate functions for **Box<int>** and **Box<float>** methods and constructors using the **C++ name mangling** scheme

Purpose of mangling: encoding of type information in function names

- C++ Compiler generates separate functions for each template instantiation:
  - Box<int>
  - Box<float>
  - Advantage:
    - Allows compiler to produce specialized code for each concrete parameter type value
    - Retrains type information
  - Disadvantage:
    - "Code bloat", two functions per generic method in the binary instead of one

Demangled Names from Object File

```
$ clang++ -std=c++11 -c box.cc
$ nm -gU box.o | c++filt
0000000000000120 T Box<float>::setContent(float const&)
00000000000000f0 T Box<float>::Box(float const&)
0000000000000180 T Box<float>::Box(float const&)
00000000000000c0 T Box<int>::setContent(int const&)
0000000000000090 T Box<int>::Box(int const&)
0000000000000160 T Box<int>::Box(int const&)
0000000000000140 T Box<float>::getContent() const
00000000000000e0 T Box<int>::getContent() const
0000000000000000 T _main
```

Two **setContent()** functions
One for **int** one for **float**

Two identical constructors, how come?
No, different constructors …C1 and …C2 in mangled names: (needed in polymorphic scenarios)
C1: Complete Object Constructor
C2: Base-object Constructor

Two **getContent()** functions
One for **int** one for **float**

- C++ Compiler generates separate functions for each template instantiation:
  - Box<int>
  - Box<float>
- Compare with Java where there is only one function for each method of **Box** in the .class file.

Type Information is **not** Erased in C++

```
#include <iostream>
#include <typeinfo>
#include "Box.H"

int main()
{
  auto intBox = Box<int>(42);       // instantiate a box that contains ints
  intBox.setContent(1);
  int i = intBox.getContent();

  auto floatBox = Box<float>(1.23);
  floatBox.setContent(3.14);
  float f = floatBox.getContent(); // instantiate a box that contains floats

  std::cout << "typeid(intBox):   " << typeid(intBox).name() << std::endl;
  std::cout << "typeid(floatBox): " << typeid(floatBox).name() << std::endl;

  return 0;
}
```

Output:

```
typeid(intBox):   3BoxIiE
typeid(floatBox): 3BoxIfE
```

Mangled type names

Demangle the Runtime Type Information

```
#include <iostream>
#include <typeinfo>
#include <cxxabi.h>

std::string demangle(const char *name)
{
  int status = -1;
  std::unique_ptr<char, void (*)(void *)> res { abi::__cxa_demangle(name, nullptr, nullptr, &status),
                                                std::free };
  return (status == 0) ? res.get() : name;
}

int main()
{
  auto intBox = Box<int>(42);
  intBox.setContent(1);
  int i = intBox.getContent();
  auto floatBox = Box<float>(1.23);
  floatBox.setContent(3.14);
  float f = floatBox.getContent();
  std::cout << "typeid(intBox):   " << demangle(typeid(intBox).name()) << std::endl;
  std::cout << "typeid(floatBox): " << demangle(typeid(floatBox).name()) << std::endl;
  return 0;
}
```

Output:

```
typeid(intBox):   Box<int>
typeid(floatBox): Box<float>
```

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

Java Collections ~~Framework~~ Library

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS 2018

Book Chapters:

- Chapter 15: "The Java Collections Framework" in Cay S. Horstmann, "Big Java Early Objects", 5th Edition.
- Chapter 9: "Collections" in Cay S. Horstmann, "Core Java", 10th Edition.

Framework vs. Library

Definitions by Martin Fowler

- **Library**:
  A set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.
- **Framework:**
  Embodies some abstract design, with more behavior built in. In order to use it, you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.

- A framework follows the Hollywood Principle, "*don't call us, we'll call you*".

- Oracle/Sun calls it the Java Collection Framework, but should be a library.
- It is not a framework
  - No-inversion of control
  - You are not adding behavior to use it.

Outline Topic 8: Java Collections

- Lists: ArrayList, LinkedList
- Iterators
- Sets and Sorted Sets
- Maps
- Stacks, Queues and Deques

- Know how to use the essential collection interfaces and their implementations.
- How to choose the appropriate Collection class.

Java Collections Framework

- Any real-world Java application need to organize objects in groups or collections in one form or another.
- **Examples**
  - Phone Book: List of entries, ordered by city and name, allows lookup of phone number given city and name.
  - ArrayList of Measurements: A temperature reading is taken every 5 minutes and appended to the array. There is an order (time) between measurements.
  - Set of Persons attending the Christmas Dinner. No order between Persons.
  - Queue of Work (Jobs) that are supposed to be executed in the order of submission or based on an assigned priority.

- Since these are frequently occurring collection types, it they are available as part of the Java Runtime Library and do not have to re-implemented or added through external libraries.

List – Set – Stack – Queue

Figure 2 A List of Books

Figure 3 A Set of Books

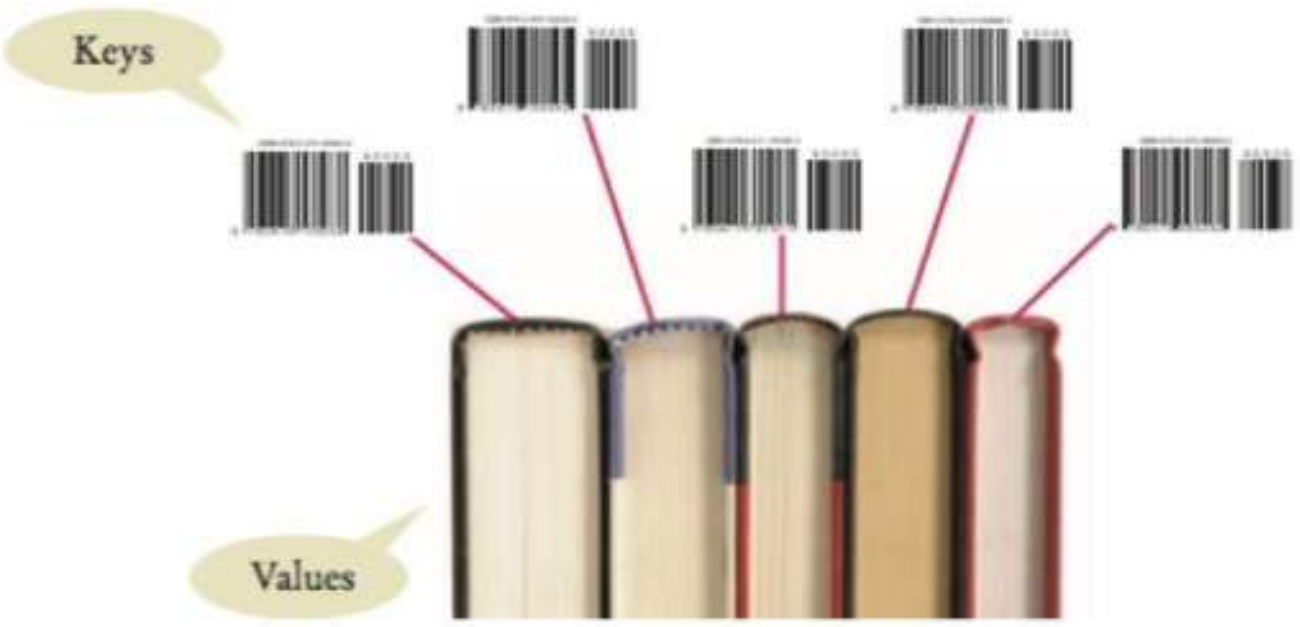
Figure 4 A Stack of Books

A Queue of People

- A List is a collection that remembers the order of the elements.
- A Set is a collection in which the elements without a particular order.
- A Stack remembers the order of the elements, elements can only be added and removed from the top of the stack. A stack is a Last-In First-Out (LIFO) data structure.
- A Queue remembers the order of the elements, elements can only be added to the tail of the queue, while elements are only removed from the head. A queue is a First-In First-Out (FIFO) data structure.
- In a Priority Queue, elements have a priority value associated to them. Elements are removed from the queue with increasing priority.

Map (Dictionary, Associative Array)

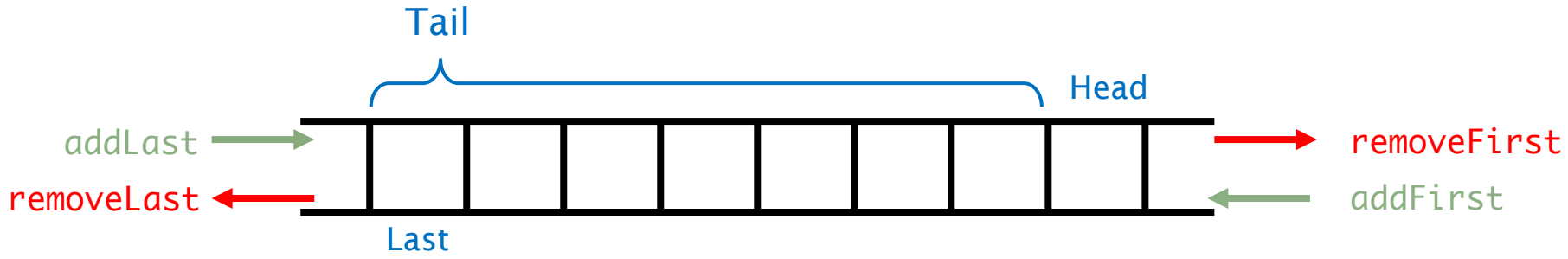
- A Map is an association between a key and value.
- A Map stores (key, value) pairs.
- Every key in a map has an associated value.
- Values can be looked up by their corresponding key.



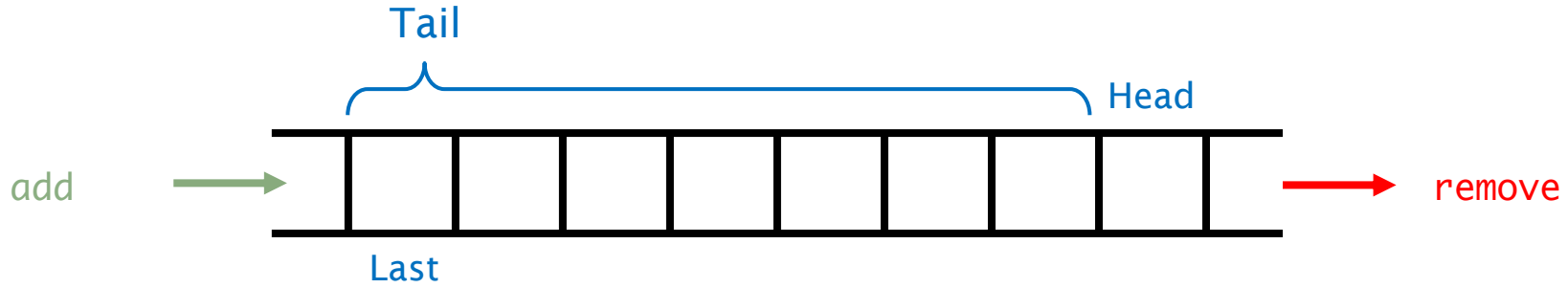
- The terms Dictionary or Associative Array are sometimes used as synonym for a Map.
- Do not confuse "a map" (noun) with "to map" (verb). Map is used in functional programming (see later).

Deque

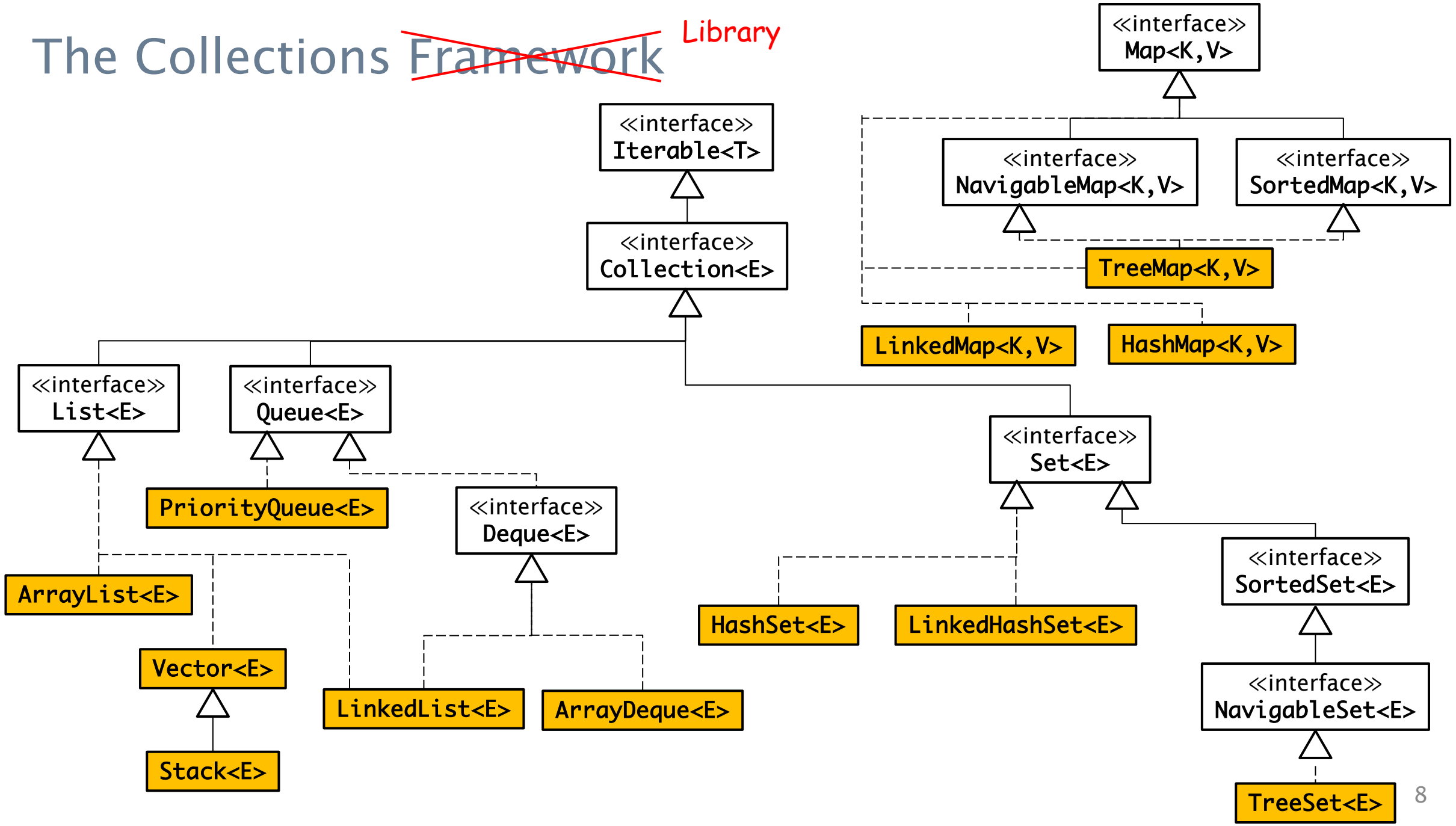
- Deque (pronunciation "deck") is a Double-Ended Queue
- **Deque:**
  - Add elements to head (`addFirst`) and tail (`addLast`)
  - Remove elements from head (`removeFirst`) and tail (`removeLast`)



- **Queue:**
  - Add new elements to tail (`add`) only
  - Remove elements from head (`remove`) only



The Collections ~~Framework~~ Library



Collections ~~Framework~~ Library

- The Java Collection Framework (JCF) provides number of interfaces and implementations (classes) of commonly used data structures. Package: `java.util`
- The JCF exists since Java 1.2 (without generics)
- Generics were added later (in Java 5)

- The generic interface `Collection<E>` is the root of the JCF.
- The interfaces `Map<K,V>` and `SortedMap<K,V>` do not inherit from `Collection<E>`, but are still members of the JCF.

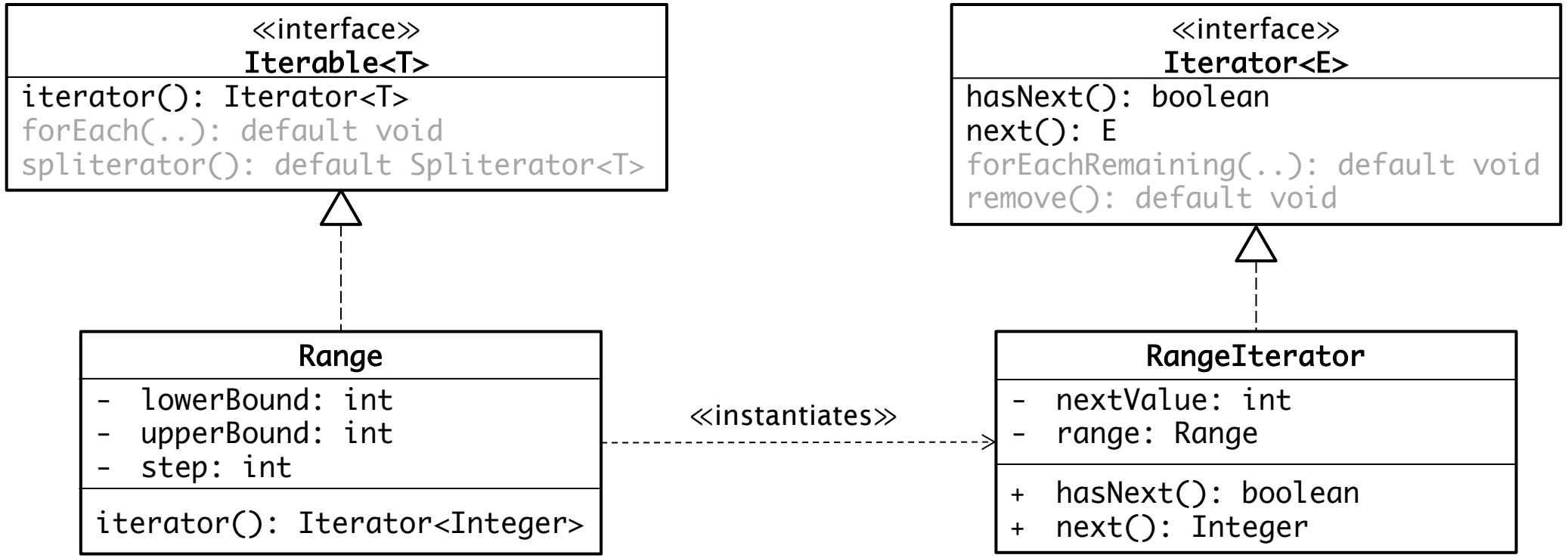
- The JCF separates the interfaces from the implementations (classes).
  - There are different implementations of an interface.
  - Trade-offs in the implementations of data structures (see BTI7062 "Algorithmen und Datenstrukturen").
    - Time vs. Space Efficiency
    - Optimization for inserts vs. lookup, point-lookup vs. traversal, etc.
- This class: *How to choose* the right implementation from the JCF.
- BTI7062 "Algorithmen und Datenstrukturen": *How to implement* the data structures yourself.

Iterable

Iterables and Iterators

Iterables and Iterators

- An `Iterator` is an object that allows traversing an `Iterable`.
  - **Iterable**: the object to be traversed (e.g., a collection, a container)
  - **Iterator**: returns object by object of the iterable until there are no more objects. The iterator is exhausted.



Using Iterators in a While-Loop

- An iterator can be used in a while loop:
  - Call `hasNext()` method to check if there more elements.
  - Call `next()` to retrieve the next element from the iterator.
  - If you call `next()` despite `hasNext()` returns `false`, you will get a `NoSuchElementException`.

- **Example**

```
Range range = new Range(1, 5);

// Iterator in while loop
Iterator<Integer> rangeIter = range.iterator();
while (rangeIter.hasNext()) {
  System.out.println(rangeIter.next());
}
```

Example: Range – Object over a Range of Integers (1)

```
import java.util.Iterator;

public class Range implements Iterable<Integer> {
  private final int lowerBound;
  private final int upperBound;
  private final int step;

  public Range(int lower, int upper, int step) {
    this.lowerBound = lower;
    this.upperBound = upper;
    this.step = step;
  }

  public Range(int lower, int upper) { this(lower, upper, 1); }

  private class RangeIterator implements Iterator<Integer> {
      // ... see next slide ...
  }

  @Override
  public Iterator<Integer> iterator() {
    return new RangeIterator();
  }
}
```

← **Range** is an Iterable of `Integer`

← **RangeIterator** is the iterator that is being returned each time the range is to be traversed. The iterator returns `Integer` objects.

Example: Range – Object over a Range of Integers (2)

```
// ... see previous slide ...

private class RangeIterator implements Iterator<Integer> {

  private int nextValue;

  RangeIterator() { nextValue = lowerBound; }

  @Override
  public boolean hasNext() {
    return nextValue <= upperBound;
  }

  @Override
  public Integer next() {
    int result = nextValue;
    nextValue = nextValue + step;
    return result;
  }
}

// ... see previous slide ...
```

← Iterator object keeps track where it is in the range.

Here, `RangeIterator` is implemented as a **private (non-static) nested class**. As such, it has access to instance variables `lowerBound`, `upperBound`, and `step` of its enclosing class.

Of course, it is always possible to implement the Iterator as a non-nested class.

Very often, the Iterator is implemented as an **anonymous inner class** in the iterable itself:

```
public Iterator<Integer> iterator() {
  return new Iterator<Integer>() {
    int nextValue = lowerBound;
    @Override
    public boolean hasNext() {
      return nextValue <= upperBound;
    }
    @Override
    public Integer next() {
      int result = nextValue;
      nextValue += step;
      return result;
    }
  };
}
```

Using Iterators in a For-Loop

```
Range range = new Range(1, 5);

// Iterator in for loop
for (Iterator<Integer> iter = range.iterator();
     iter.hasNext(); ) {
  System.out.println(iter.next());
}
```

empty update clause.

continuation condition

Using Iterables in For-Each Statement

Available since Java 5

- You can specify an iterable (not an iterator!) in a for-each statement.
- An iterator will be obtained from the iterable at the beginning of the loop.

```
Range range = new Range(1, 5);

// Iterable in foreach loop
for (Integer i : range) {
  System.out.println(i);
}
```

Specify the iterable object, not the iterator.

Using forEach Method of Iterables

Available since Java 8

- In Java 8, the forEach(..) Method was added to the Interface **Iterable**. The interface provides also a suitable default implementation.

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

- It can be used to traverse the iterable while applying an action (without result) on each element.

Example:

```
Range range = new Range(1, 5);

// Iterator in forEach with explicit anonymous Consumer implementation.
range.forEach(new Consumer<Integer>() {
  @Override
  public void accept(Integer integer) {
    System.out.println(integer);
  }
});
```

Invoking forEach Method with Lambda Expression

Available since Java 8

- Java 8 also introduced functional interfaces. Interfaces with one single method that are annotated as `@FunctionalInterface` can be targeted by lambda expressions and method references.

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    ...
}
```

Example:

```
Range range = new Range(1, 5);

// Iterator in forEach using Consumer as a functional interface with lambda.
range.forEach(i -> System.out.println(i));

// Iterator in forEach with explicit anonymous Consumer implementation.
range.forEach(new Consumer<Integer>() {
  @Override
  public void accept(Integer integer) {
    System.out.println(integer);
  }
});
```

equivalent, but less typing required.

This lambda can take an Integer and returns void, thus, it **is-a** `Consumer<Integer>`.

Invoking forEach Method with Method Reference

Available since Java 8

- Java 8 also introduced functional interfaces. Interfaces with one single method that are annotated as `@FunctionalInterface` can be targeted by lambda expressions and method references.

Example: 3 Equivalent ways to use `forEach(..)`

```
Range range = new Range(1, 5);

// Iterator in forEach using Consumer as a functional interface with a
// method reference.
range.forEach(System.out::println);

// Iterator in forEach using Consumer as a functional interface with lambda.
range.forEach(i -> System.out.println(i));

// Iterator in forEach with explicit anonymous  Consumer implementation.
range.forEach(new Consumer<Integer>() {
  @Override
  public void accept(Integer integer) {
    System.out.println(integer);
  }
});
```

equivalent, but even less typing required.

The method `println()` can take an Integer and returns void, thus, it **is also a** `Consumer<Integer>`.

Collection Interface

`Collection<E>` Interface

- A collection represents an iterable group of elements.
- There are various types of collections, depending on how the elements are stored and on how manipulations work:
  - Some implementations allow duplicate elements, others do not.
  - Some are ordered, others are unordered.
  - Some have a fixed (or maximal) size, others have a variable size.
  - Some can be modified, others are immutable.

Methods of the `Collection<E>` Interface

Most important methods, for complete list see API Documentation https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Method	Description
`boolean add(E e)`	Ensure that collection contains element e (returns `true` if added)
`boolean addAll(Collection<? extends E> c)`	Adds all elements from collection c to this collection (returns `true` if this collection changed as a result of the call).
`boolean remove(E e)`	Removes single element from collection (returns `true` if present)
`boolean removeAll(Collection<? extends E> c)`	Remove all elements that are in c from this collection (return `true` if this collection changed as a result of the call).
`int size()`	Returns the number of elements in the collection.
`void clear()`	Removes all elements from the collection.
`boolean isEmpty()`	Returns `true` if the collection is empty.
`boolean contains(Object o)`	Returns `true` if the collection contains the specified element. Uses `equals()` method of element type.
`Iterator<E> iterator()`	Returns an iterator over the elements in the collection.
`Object[] toArray()`	Convert collection to a new Java array. Returns an object array.
`E[] toArray(E[] array)`	Convert collection to array. Creates new array of same dynamic type as argument if provided array not large enough. Returns array of element type.
`String toString()`	Returns a string representation of the collection and its elements.

Extensions of `Collection` Interface

- `List<E>`: Ordered Collection (Sequence) with arbitrary access
  - User has precise control over where the elements are stored.
  - Elements are accessed by their integer index.
  - Allow duplicate elements.
- `Queue<E>`: Ordered Collection with FIFO access
- `Deque<E>`: Ordered Collection that supports insert and access on both ends

- `Set<E>`: Unordered Collection that does not contain duplicate elements (a mathematical set).
  - Iterator will traverse set in an unspecified order.
- `SortedSet<E>`: A Set with a total order. Iterator will traverse set in ascending order.
- `NavigableSet<E>`: A sorted set with navigation methods that report the closest match to search element.

Classes of the Collections Library

- A class that implements `Collection<E>` is supposed to provide at least two constructors for constructing
  - an empty collection.
  - a collection containing all elements of another collection.

- **Examples:**
  - `ArrayList(), ArrayList(Collection<? extends E> c)`
  - `LinkedList(), LinkedList(Collection<? extends E> c)`
  - `HashSet(), HashSet(Collection<? extends E> c)`
  - etc.

https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Collection Interface and the Classes Implementing It

Collection (Java Platform SE 8 ×

Secure https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Interface Collection<E>

Type Parameters:

E - the type of elements in this collection

All Superinterfaces:

Iterable<E>

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

```
public interface Collection<E>
extends Iterable<E>
```

There are 34 classes in the Java 8 standard library that implement `Collection<E>`.
Here, we focus only on the few most important.

Example: Using Collections

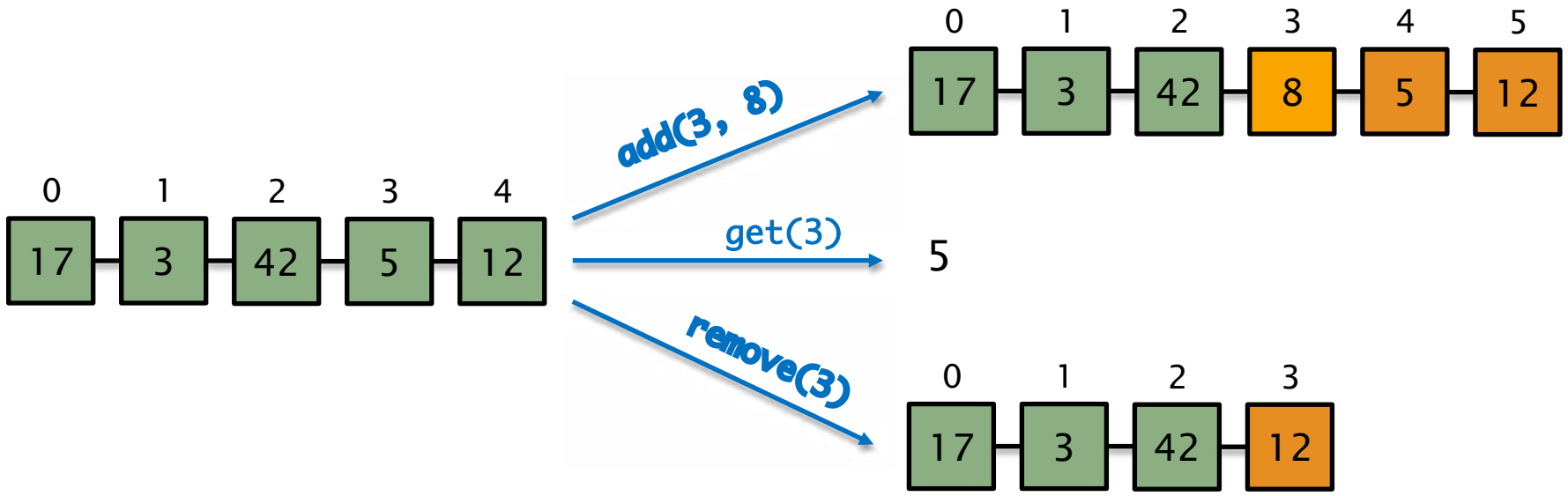
```
public class CollectionsExample {

  public static void main(String[] args) {
    Collection<String> coll1 = new ArrayList<>();
    coll1.add("Hello");
    coll1.add("World");
    coll1.size();        // returns 2
    System.out.println(coll1); // prints "[Hello, World]"

    Collection<String> coll2 = new LinkedList<>(coll1);
    coll2.add("!");
    coll2.size();        // returns 3
    coll2.remove("World");
    coll2.size();        // returns 2
    System.out.println(coll2);  // prints "[Hello, !]"

    Collection<String> coll3 = new HashSet<>(coll2);
    coll3.add("World");
    coll3.add("World");
    coll3.size();      // returns 3
    System.out.println(coll3); // prints "[!, Hello, World]"
  }
}
```

Lists

ArrayList, LinkedList,
Vector, and Stack

Lists

- Lists are ordered collections.
- A list offers positional access to the elements through an integer index.
  - Elements can be inserted at a specific location.
  - Elements at specific location can be retrieved.
  - Elements at specific location can be removed.



Recap OOP1: Regular Java Arrays

- Regular Java Arrays, e.g., `int[]` or `double[]` are part of the Java Language, not the Java Library.
- Java Arrays have a fixed size that is specified at construction time with new:

```
int[] numbers = new int[5];
```

- Java Arrays cannot be extended or shortened.
- Java Arrays do not have any methods.
- Java Arrays have a single read-only attribute `.length` that returns then length (capacity) of the array.

- Java Arrays are covariant:
  - `SavingsAccount[]` is a subtype of `BankAccount[]` where `SavingsAccount` is a subtype of `BankAccount`.

Recap OOP2: `ArrayList`

- We used `ArrayList` as a "better" version of plain Java arrays.
  - `ArrayLists` can be extended (and shortened).
  - `ArrayList` provide additional useful methods (e.g., `toString()`).

- `ArrayList` is a generic class. It is parameterized by a type of the elements it can store.
  → It can store objects of an arbitrary class (but no values of primitive data types).

- `java.util.ArrayList` is a class that implements the `List<E>` contains an array-based list implementation (see https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

Methods of the List<E> Interface

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

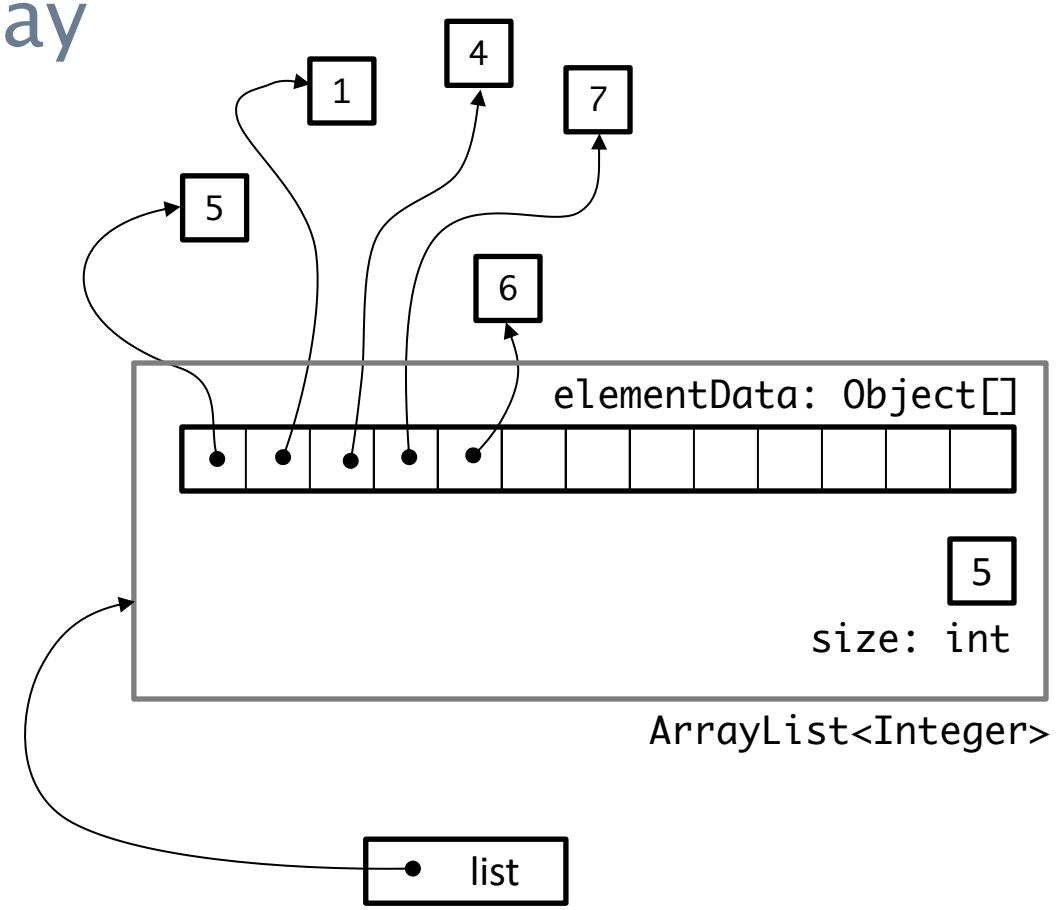
The List<E> Interface extends the Collection<E> Interface.

Methods	Description
boolean add(E e)	Append element to the end of the list (always returns true)
void add(int index, E e)	Insert element at specified position (shift elements to the right)
boolean add(int index, Collection<? extends E> c)	Inserts the elements of c into this list starting from the specified position (returns true if the list changed as a result).
E get(int index)	Returns the element at the specified position.
int indexOf(Object o)	Returns the index of the first occurrence of e in the list, returns -1 if the e is not in the list (uses equals()).
int lastIndexOf(Object o)	Returns the index of the last occurrence of e in the list, returns -1 if the e is not in the list (uses equals()).
E set(int index, E e)	Replaces the element at the specified position.
E remove(int index)	Removes single element at specified position and returns it.
E remove(Object o)	Removes first occurrence of object and returns it (uses equals())

ArrayList: List Backed by an Array

- The content (object references) of **ArrayList** object is stored in a Java array.
- The filed specifies which array entries contain valid content.
  - `elementData[i]` is valid for `i = 0 .. size-1`.

- Since Java arrays cannot grow or shrink, a new arrays must be created when the `elementData` array is full. The content need to be copied over to the new array.
  - In Java 8, the array grows by 50%.
  - Note that removals do not shrink the array. The array must be shrunk explicitly to `size` by calling `trimToSize()` method.

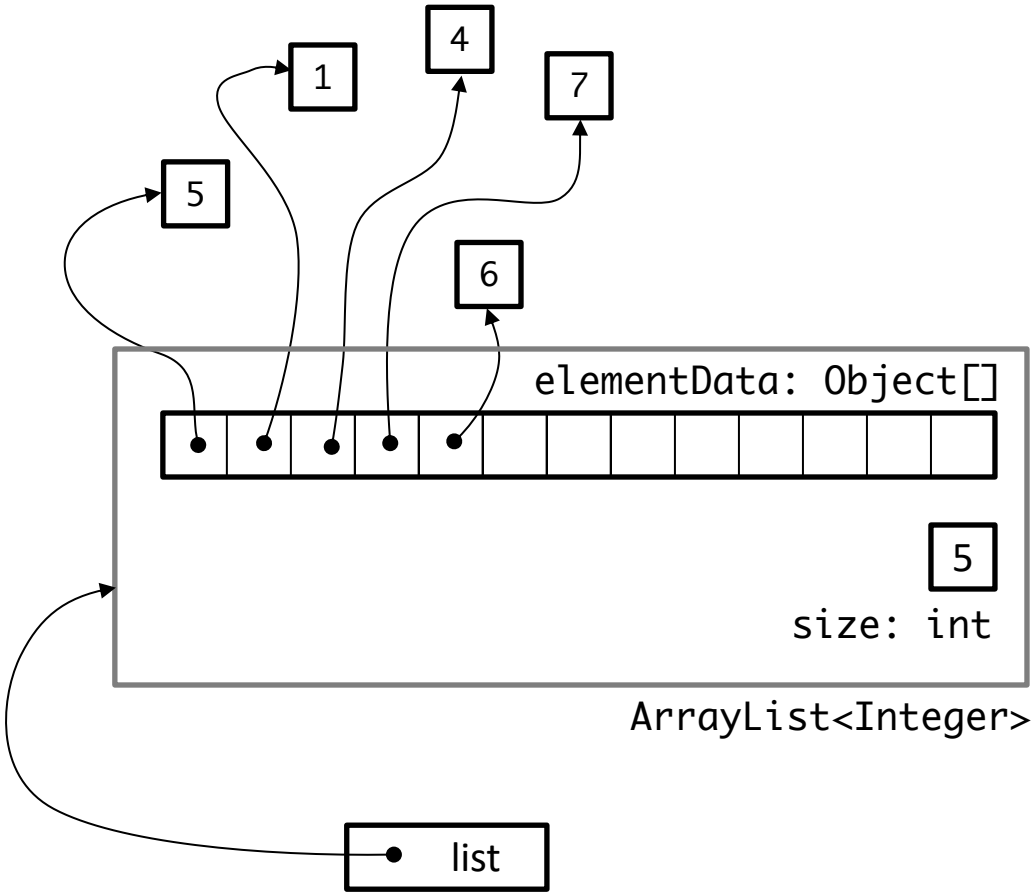



```
List<Integer> list = new ArrayList<>();
list.add(5);
list.add(1);
list.add(4);
list.add(7);
list.add(6);
```

Inserts in `ArrayList` (1)

- When inserting an element into an array list, the existing elements need to be shifted to the right.
- A single insert at a random position in an `ArrayList` with *n* elements requires moving of *n*/2 elements on average!

- When deleting an element in the middle, the gap needs to be closed by shifting the elements right of the delete position to the left.
  → also n/2 elements need to be moved on average.

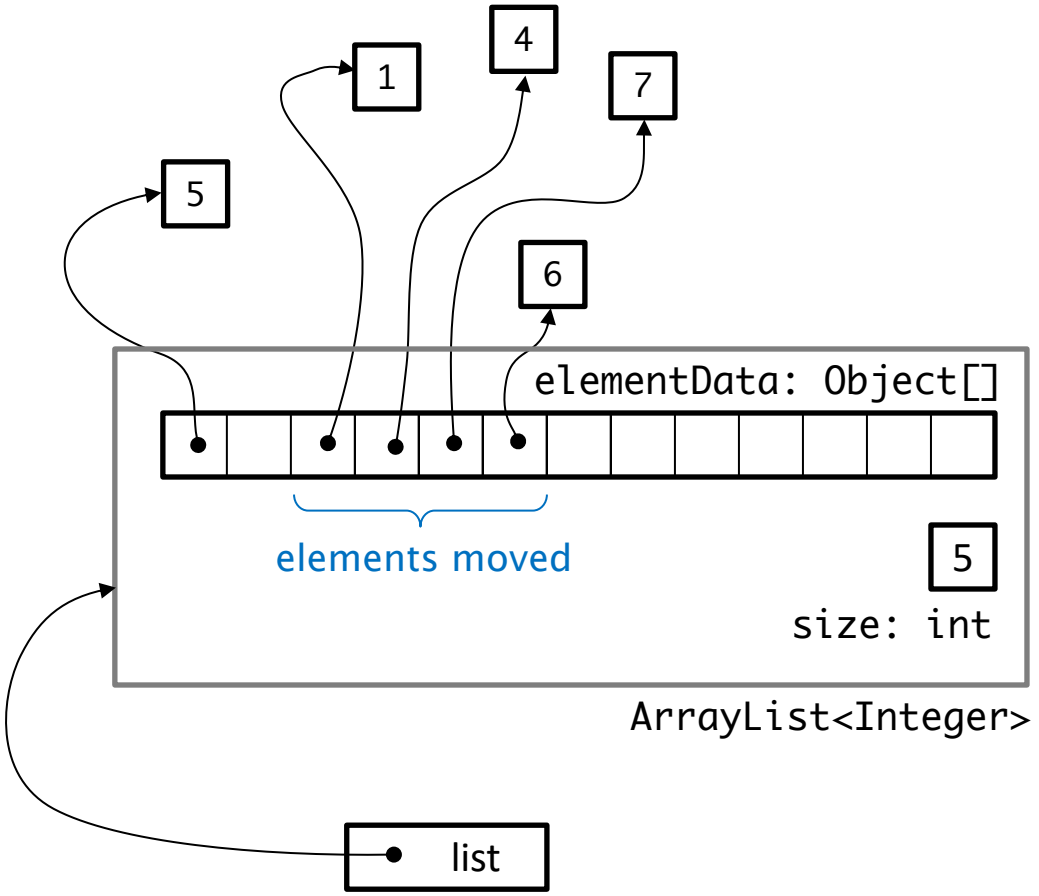



```
list.add(1, 8);
```

Inserts in `ArrayList` (2)

- When inserting an element into an array list, the existing elements need to be shifted to the right.
- A single insert at a random position in an `ArrayList` with *n* elements requires moving of *n*/2 elements on average!

- When deleting an element in the middle, the gap needs to be closed by shifting the elements right of the delete position to the left.
  → also n/2 elements need to be moved on average.

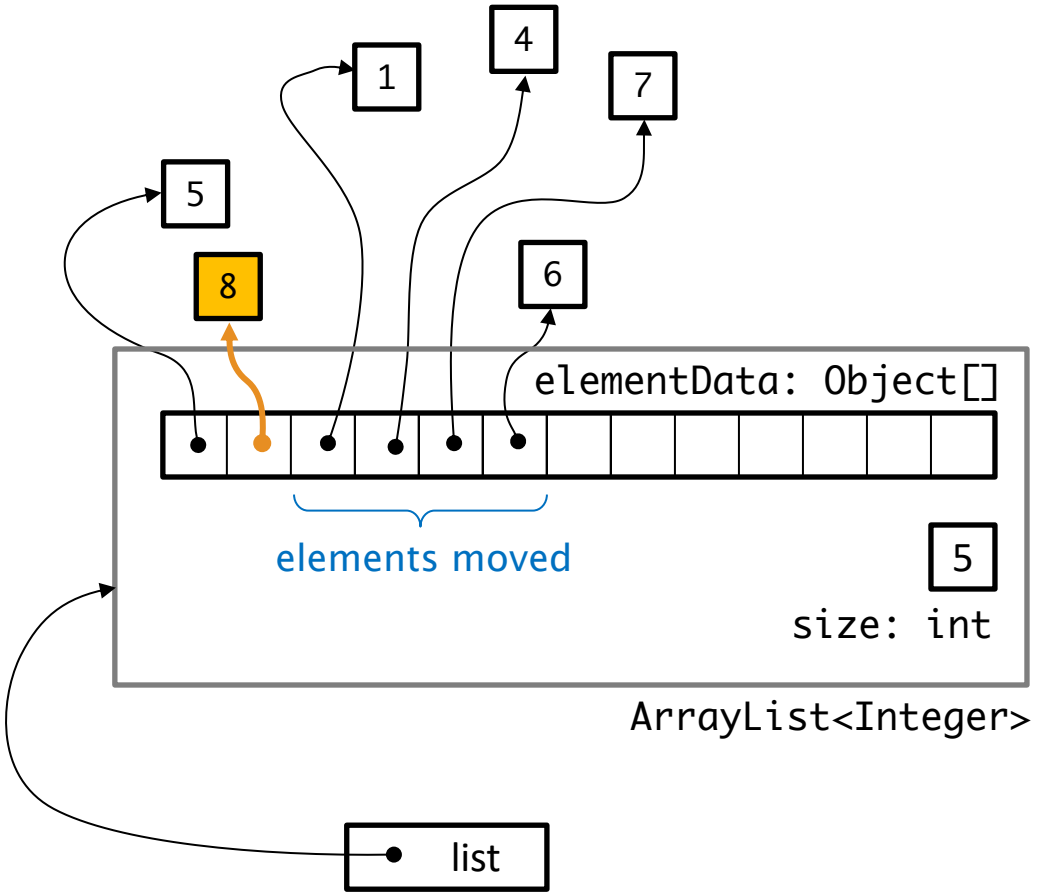



```
list.add(1, 8);
```

Inserts in `ArrayList` (3)

- When inserting an element into an array list, the existing elements need to be shifted to the right.
- A single insert at a random position in an `ArrayList` with *n* elements requires moving of *n*/2 elements on average!

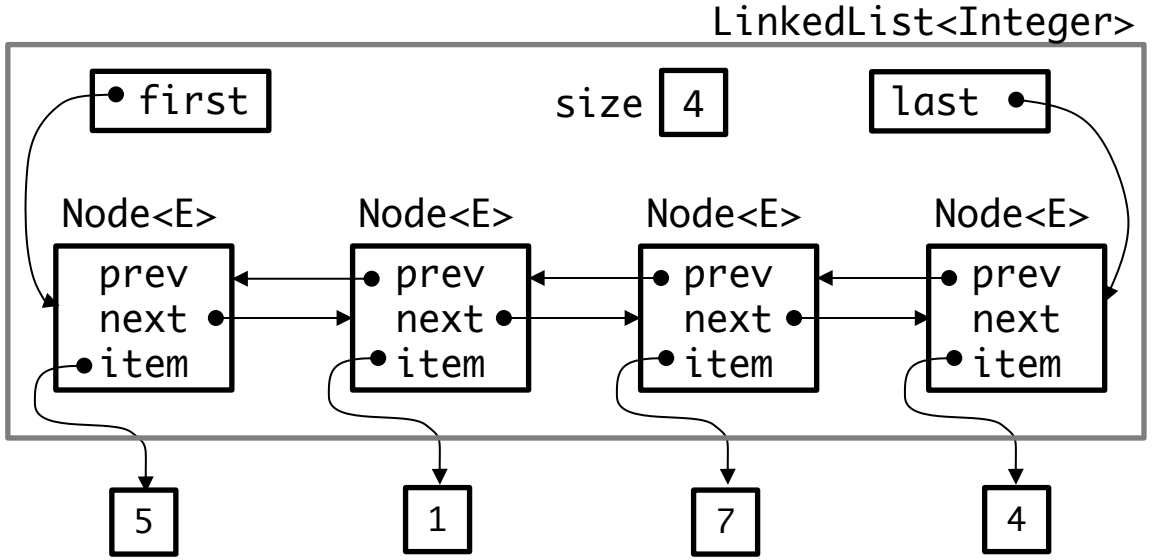
- When deleting an element in the middle, the gap needs to be closed by shifting the elements right of the delete position to the left.
  → also n/2 elements need to be moved on average.




```
list.add(1, 8);
```

LinkedList

Covered extensively in BTI7062 "Algorithmen und Datenstrukturen"



- **LinkedLists** based on doubly-linked lists.
  - The lists consists of Nodes.
  - Every node has a link **next** to the next node.
  - Every node has a link **prev** to the previous node.
  - Every node has a reference **item** to the payload, i.e., the stored object.
- The first and the last node in the list are also reachable directly by references **first** and **last**.

```
list.add(1, 8);
```

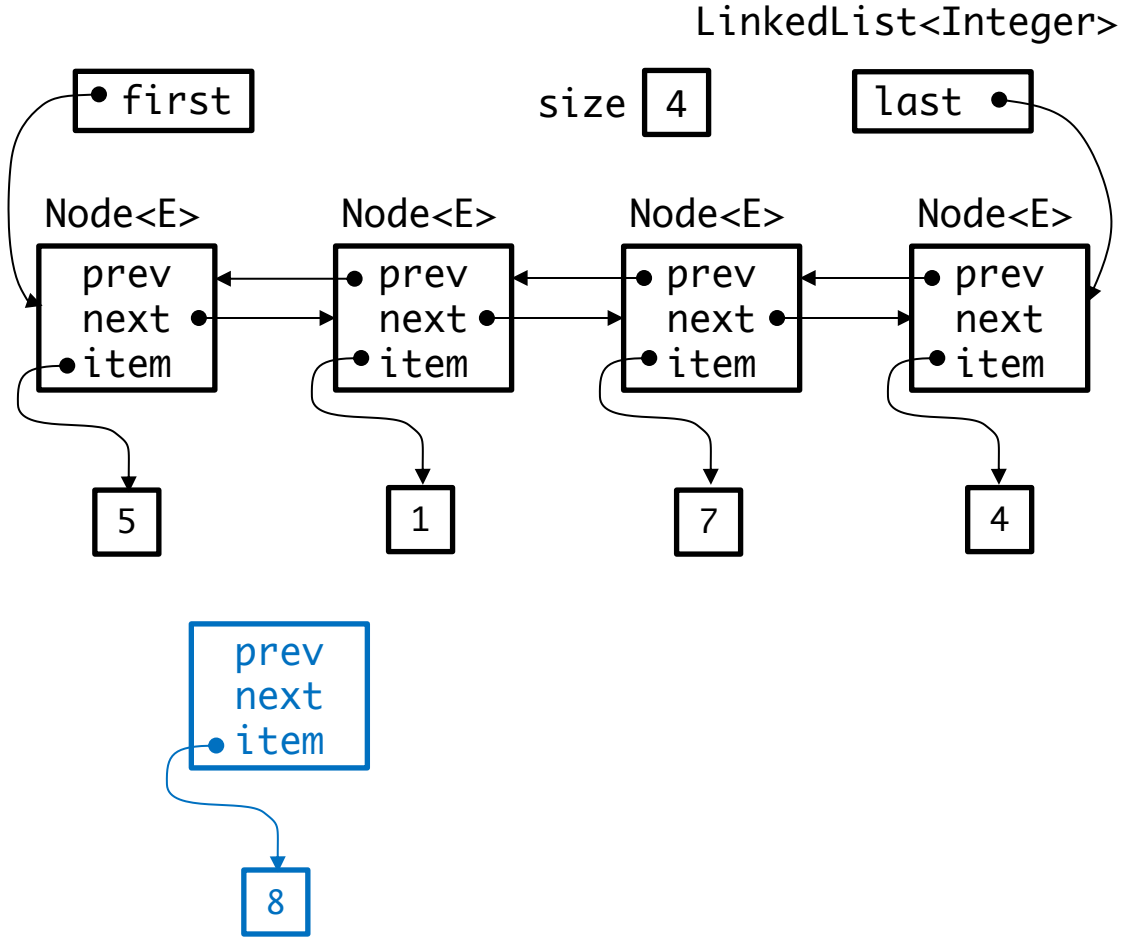
- An individual node can be accessed by following the links.
  - Forward from **first** following the **next** links.
  - Backward from **last** following the **prev** links.
  - Accessing a random node in a list of n nodes requires $n/4$ link traversals on average.
- Inserting a new node simply requires modifying the **next** and **prev** links.

Covered extensively in BTI7062 "Algorithmen und Datenstrukturen"

Insert into LinkedList (1)

```
list.add(1, 8);
```

1. Create new Node object.

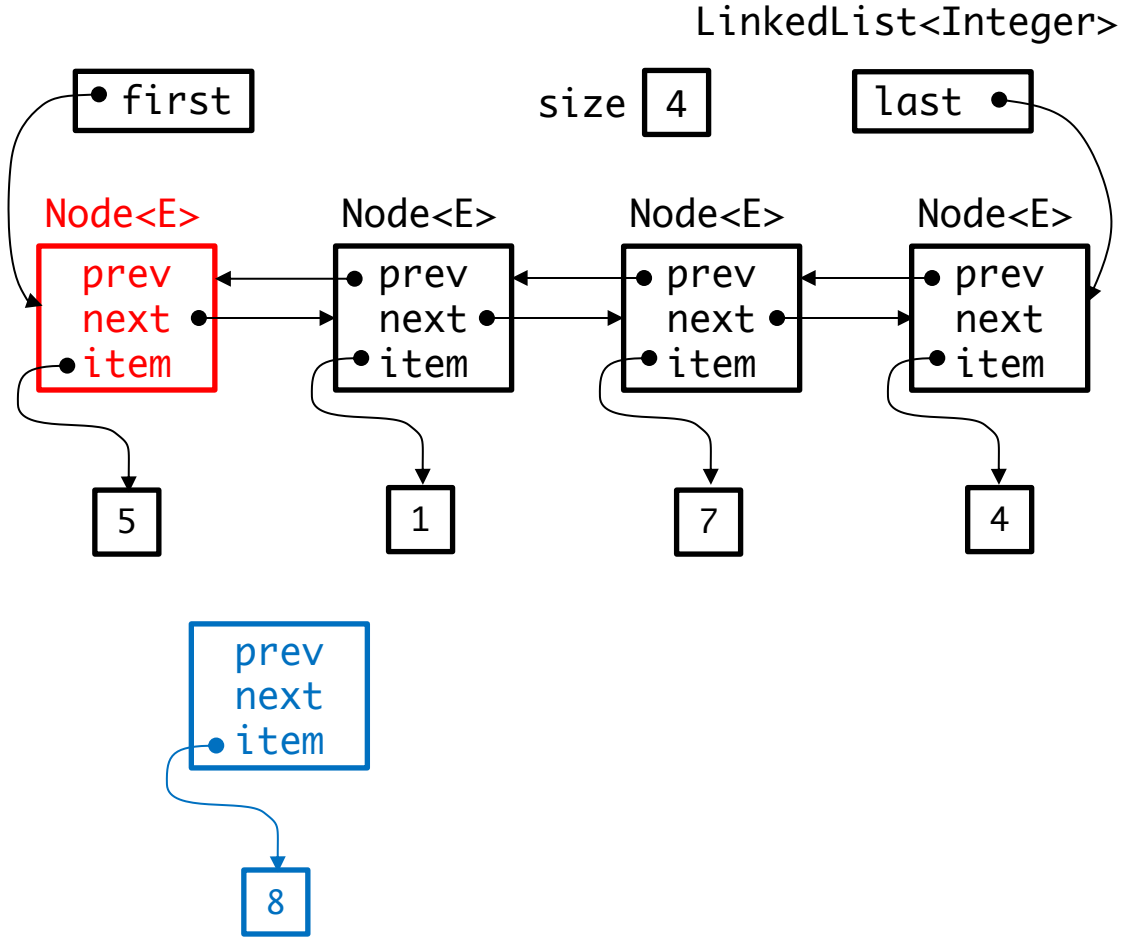


Covered extensively in BTI7062 "Algorithmen und Datenstrukturen"

Insert into LinkedList (2)

```
list.add(1, 8);
```

1. Create new Node object.
2. Find insertion place (after red node).

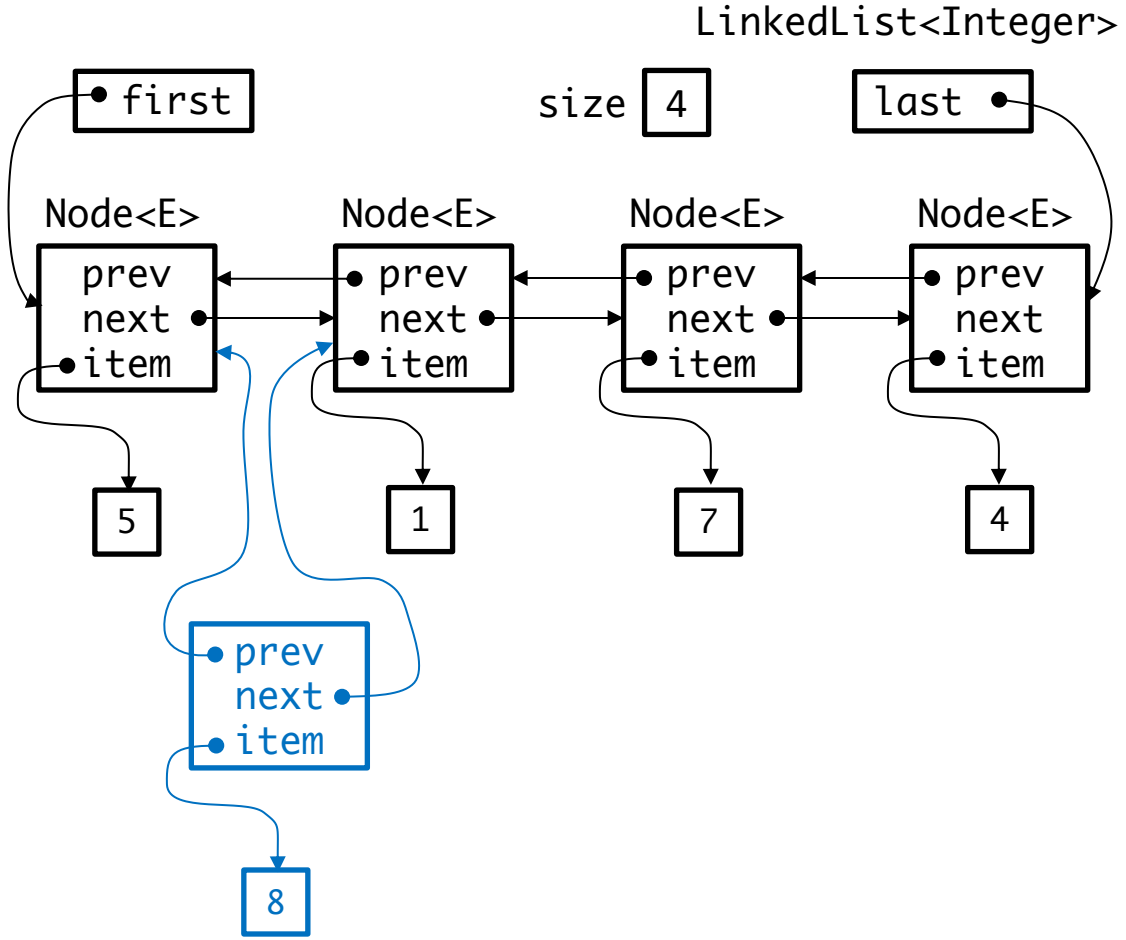


Covered extensively in BTI7062 "Algorithmen und Datenstrukturen"

Insert into LinkedList (3)

```
list.add(1, 8);
```

1. Create new Node object.
2. Find insertion place (after red node).
3. Set **next** and **prev** links of new Node object.

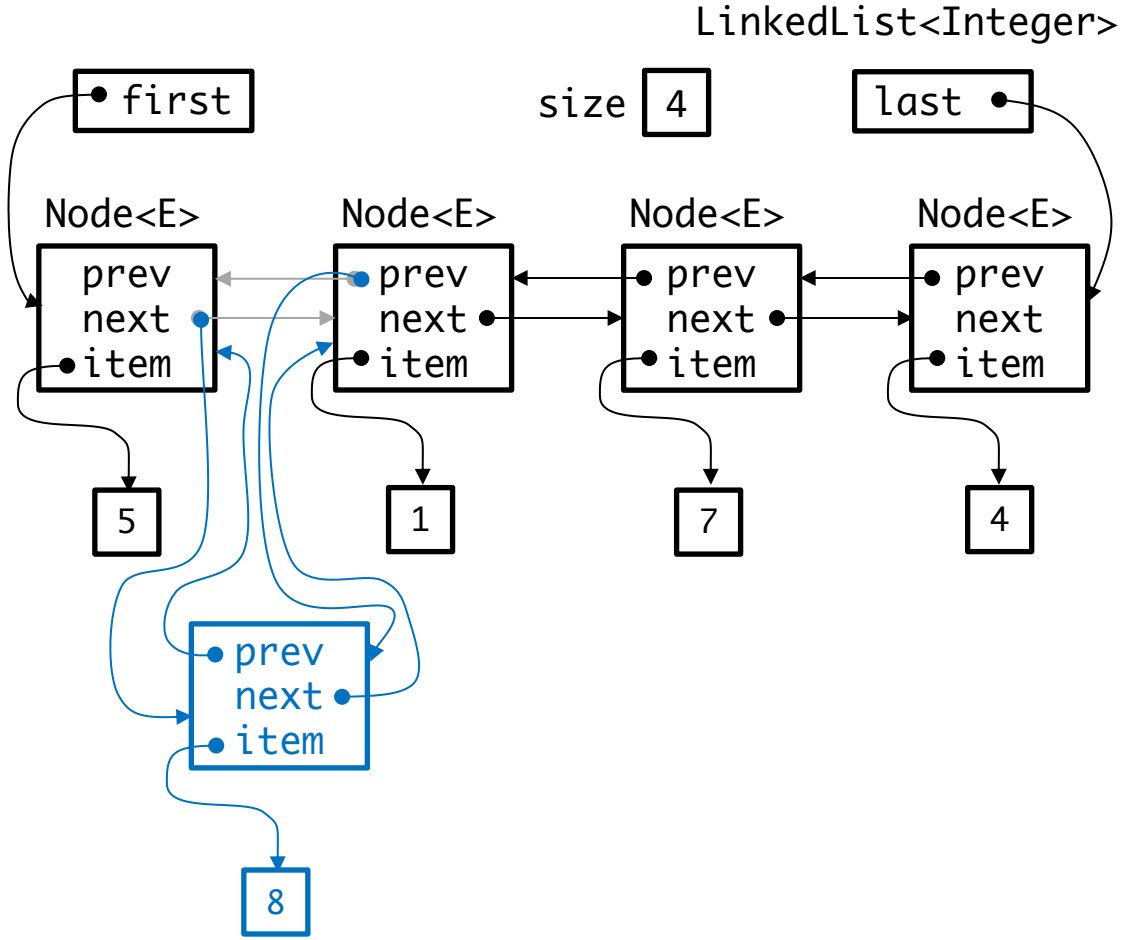


Covered extensively in BTI7062 "Algorithmen und Datenstrukturen"

Insert into LinkedList (4)

```
list.add(1, 8);
```

1. Create new Node object.
2. Find insertion place (after red node).
3. Set **next** and **prev** links of new Node object.
4. Modify **next** and **prev** links of old Node objects.



ArrayList vs. LinkedList

Recommendation: `ArrayList` if mostly `get()` in your program, otherwise, use `LinkedList`.

Each List implementation has its advantages and disadvantages. → Choose based on application needs.

	ArrayList	LinkedList
Insert Element to End (append)	[ ] Usually simple if array has capacity, otherwise, a new array needs to be created and the old entries copied.	[+] Simple, just add new node to `last` and update links.
Insert Element at the Front	[-] Expensive, need to move n elements, and potentially create larger array.	[+] Simple, just add new node to `first` and update links.
Insert Element in the Middle	[-] Expensive, need to move n/2 elements on average, and potentially enlarge array	[ ] Follow links until insertion point found, add new node, update links.
Access i^{th} Element	[+] Simple, direct index access	[-] Expensive, follow `i` links.
Remove Element at the End	[+] Simple (no shrinking)	[+] Simple, just update links.
Remove Element at the Front	[-] Expensive, need to move n-1 elements (no shrinking).	[+] Simple, just update links.
Remove Element in the Middle	[-] Expensive, need to move n/2 elements on average (no shrinking).	[ ] Follow links until deletion point found, update links.

Lists – Performance Comparison

Output

```
ArrayList: 21.425176465 sec
LinkedList: 0.020979689 sec
```

→ ArrayList is 1,000× faster!

```java
public class ListTest {

  public static void main(String[] args) {
    final int rounds = 500000;

    List<String> arrayList = new ArrayList<>();
    long startTime = System.nanoTime();
    for (int i = 0; i < rounds; i++) {
      arrayList.add(0, "Hello");  // insert to front
    }
    long endTime = System.nanoTime();
    System.out.println("ArrayList: "+ ((endTime-startTime)*1e-9) + " sec");

    List<String> linkedList = new LinkedList<>();
    startTime = System.nanoTime();
    for (int i = 0; i < rounds; i++) {
      linkedList.add(0, "Hello"); // insert to front
    }
    endTime = System.nanoTime();
    System.out.println("LinkedList: "+ ((endTime-startTime)*1e-9) + " sec");
  }
}
```

Measure how long it takes to insert 500,000 elements to the front of an ArrayList.

Measure how long it takes to insert 500,000 elements to the front of a LinkedList.

Vector

- **`Vector`** is a growable array of objects that can be accessed over an index.
- It is the oldest Collection class the even predates the Java Collection Framework (JCF).
- Vector was retrofitted into JCF to fit the **`List<E>`** Interface.

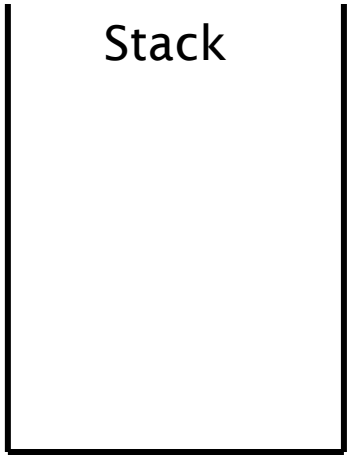
- Unlike the other "new" classes of the JCF, **`Vector`** thread-safe (i.e., synchronized).
- Unless you need thread-safe implementation of **`List`**, use **`ArrayList`** instead.
- For multithreading applications, there are also classes in **`java.util.concurrent`**.

Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

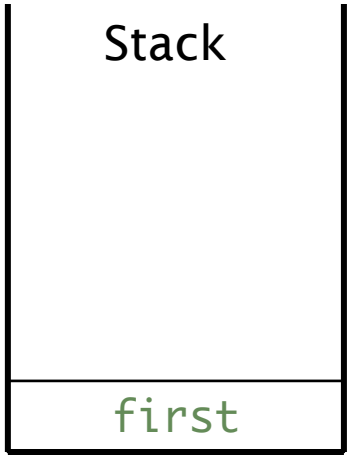


Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

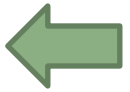


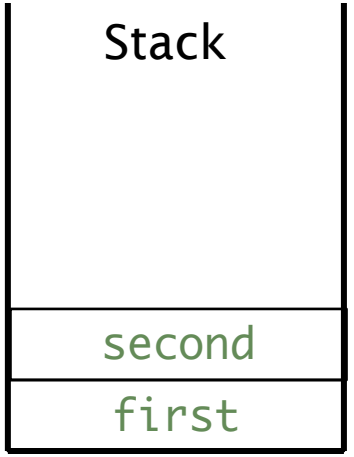
Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```



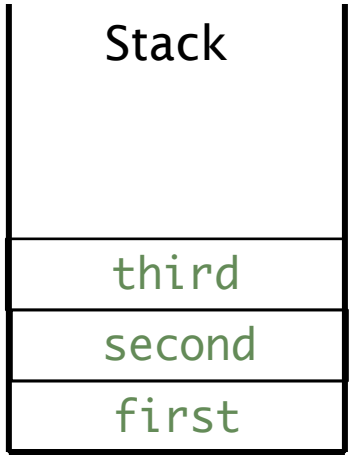


Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

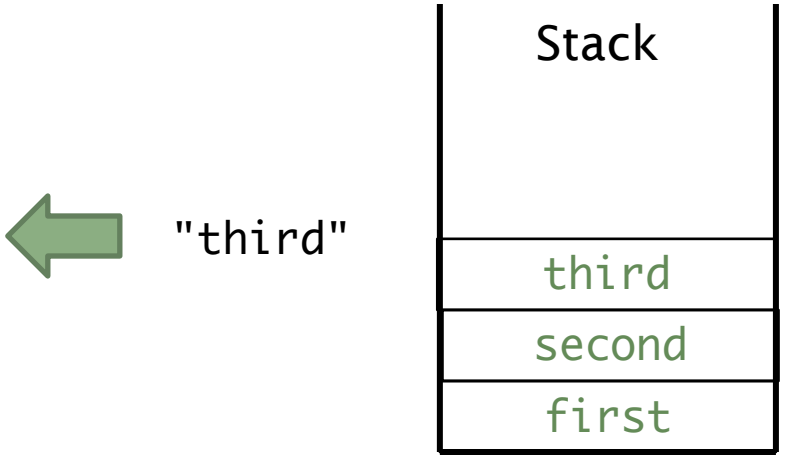


Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

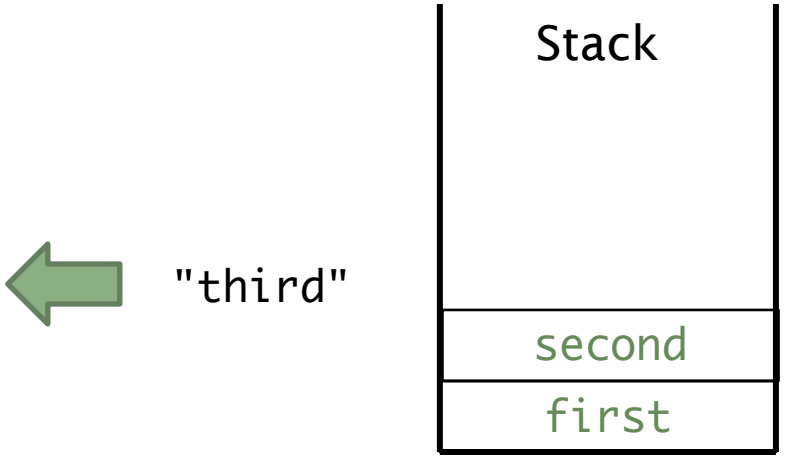


Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

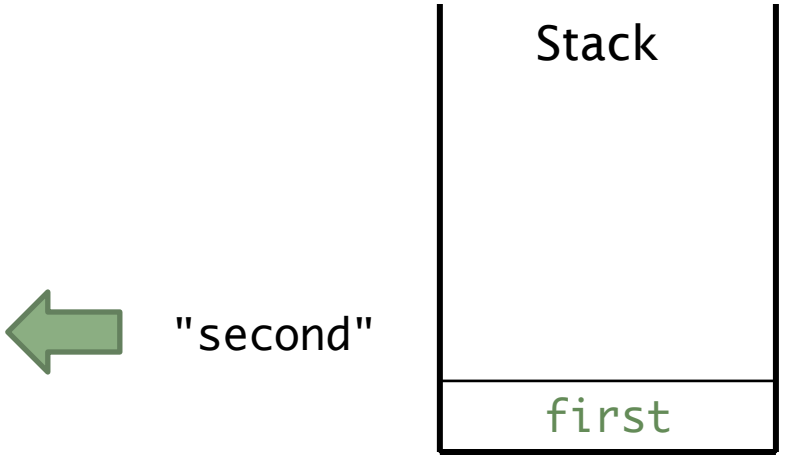


Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

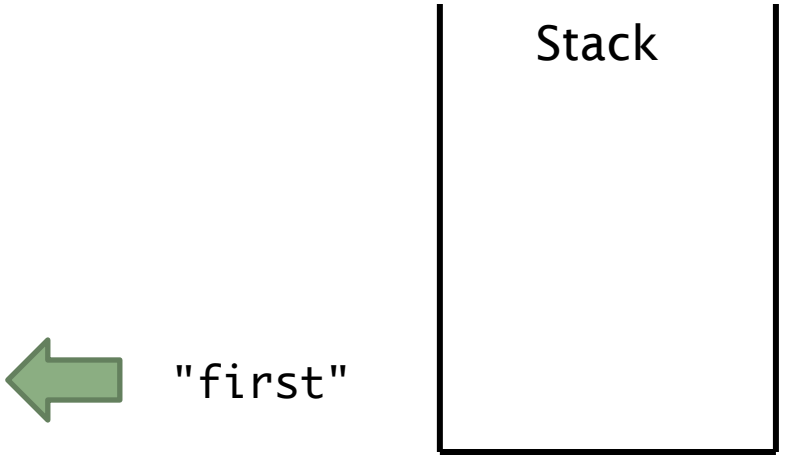


Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

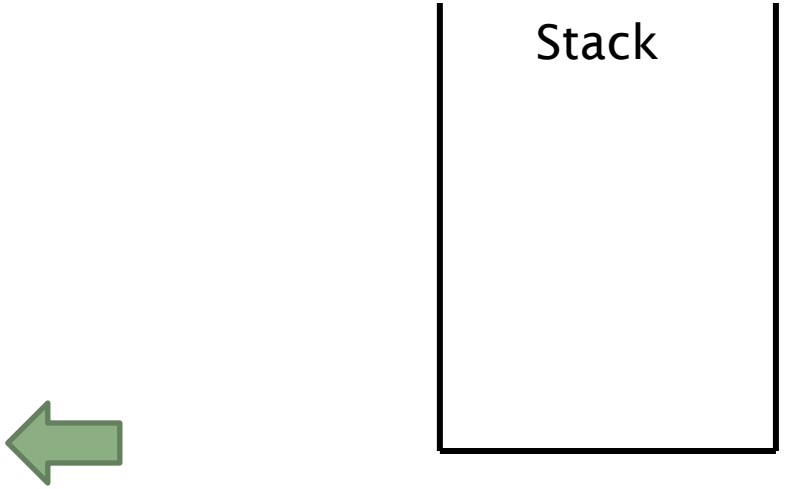


Stack

- A Stack offers Last-In, First-Out (LIFO) processing of elements, like a stack of cards or a stack inside a CPU.

- **Example**:

```
Stack<String> stack = new Stack<>();
stack.push("first");
stack.push("second");
stack.push("third");
System.out.println(stack.peek());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```




```
Exception in thread "main"
java.util.EmptyStackException
```

See https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

Stack<E> Class

A Stack<E> is a subclass of Vector<E>, which in turn is a List<E>.
In addition to the inherited methods, a Stack<E> provides the following additional methods:

Method	Description
`boolean empty()`	Tests if the stack is empty.
`E peek()`	Looks at the object on top of the stack without removing it from the stack.
`E pop()`	Removes the object at the top of the stack and returns that object.
`E push(E e)`	Pushes an into onto the top of the stack. Returns the item **e** as a result.
`int search(Object o)`	Returns the 1-based position where an object is on the stack. (1 = element is on top of the stack, 2 = one element below the element on stop, etc.)

In-class Exercise

Use a Stack to Match Opening and Closing Parentheses

"( )", "[ ]", "{ }"

In-class Exercise

- Implement a class **ParenthesesMatcher** with a static method
  `public static void ensureParenthesesBalanced(String expression) { }`

- This method verifies that all `'('`, `'['`, and `'{'` are matched by the opposite parenthesis, i.e., `')'`, `']'`, and `'}'`.
- If the expression contains one or more non-matching parentheses, it throws an `IllegalArgumentException`.
  - Good expression: `"{-b + sqrt(b*b - 4 * a * c)}/[2 * a]"`, `"{[]()([]())}"`
    (only check that the parentheses match, ignore whether the expression makes sense or not)
  - Bad expression: `"([{]})"`, `"("`

- Use a Stack to keep track of the opening parentheses.
- Iterate over expression string character-by-character and only pay attention to the six characters `'('`, `'['`, `'{'` , `')'`, `']'`, and `'}'`.

- Test your implementation using the provided **ParenthesesMatcher** JUnit Test Case.
  (Download it from Moodle: https://moodle.bfh.ch/mod/resource/view.php?id=664613)

Sets

HashSet and TreeSets

Sets

- A set is a collection that does not have any duplicates.

- Sets differ from lists by two aspects:
  - A list may contain duplicates, a set does not.
  - The elements in a list are ordered. Positional access over an index has a meaning. Sets are unordered, in general, like mathematical sets (except for `SortedSet<E>`).

- **Example**

```
Set<String> set = new HashSet<>();
set.contains("Foo");        // returns false
set.add("Foo");
set.add("Bar");
set.contains("Foo");        // returns true
set.add("Foo");
System.out.println(set);    // prints [Bar, Foo]
set.remove("Bar");
set.contains("Bar");        // returns false
```

← "Bar" appears before "Foo" even though "Foo" was added first.

See https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

Set<E> Interface

The interface `Set<E>` extends the interface `Collection<E>`.

Methods	Description
`boolean add(E e)`	Add element to set (returns `true` if the set was not already in set). $S \leftarrow S \cup \{ e \}$
`boolean contains(Object o)`	Returns `true` if object o is in the set (uses `equals()`). $o \in S$
`boolean remove(Object o)`	Removes object from set (returns `true` if element was in set). $S \leftarrow S \setminus \{ o \}$
`boolean isEmpty()`	Returns `true` if set does not contain any elements. $\lvert S \rvert = 0$
`int size()`	Returns number of elements in set. $\lvert S \rvert$
`boolean add(` `Collection<? extends E> c)`	Inserts the elements of c into this set (returns `true` if set changed as a result of this call). $S \leftarrow S \cup Set(c)$
`boolean containsAll(` `Collection<?> c)`	Returns `true` if set contains all elements from collection c. $S \leftarrow S \supseteq Set(c)$
`boolean removeAll(` `Collection<?> c)`	Removes all elements from the set that are in c (returns `true` if set changed as a result of the call.) $S \leftarrow S \setminus Set(c)$
`boolean retainAll(` `Collection<?> c)`	Removes all elements from the set **except** those that are in c (returns `true` if set changed as a result of the call.) $S \leftarrow S \cap Set(c)$

See https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html

SortedSet<E> Interface

- A sorted set in Java is a set that provides a total order on its elements, which is determined by their natural order (as determined the implementation of `Comparable<E>` of the elements' class) or by specifying a comparator object when instantiating the set.

- The interface `SortedSet` provides some additional methods that exploit the existence of an order.

Addition Methods in SortedSet	Description
`E first()`	Returns the minimal element currently in this set.
`E last()`	Returns the maximal element currently in this set.
`SortedSet<E> tailSet(E e)`	Returns elements that are greater than or equal to `e`.
`SortedSet<E> headSet(E e)`	Returns elements that are strictly less than `e`.
`SortedSet<E> subSet(E e1, E e2)`	Returns the elements that range from `e1` (inclusive) to `e2` (exclusive).

Testing for Equality of Elements in Set<E> and SortedSet<E>

- Two elements e1 and e2 in a sets are duplicates if they are equal.
- Equality in Set and SortedSet is defined differently:
  - Equality in Set: `e1.equals(e2)`
  - Equality in SortedSet using the elements natural order: `e1.compareTo(e2) == 0`
  - Equality in SortedSet using a Comparator object c: `c.compareTo(e1, e2) == 0`

- Object equality with `boolean equals(Object obj)`: (recap from OOP1)
  - Method `boolean equals(Object obj)` in class Object only checks identity `e1 == e2`
  - If you need Set<MySet>, you must override `equals(Object obj)` in MyClass.
  - Rules:
    - `e1.equals(null) == false`
    - `e1.equals(e1) == true`
    - `e1.equals(e2) == e2.equals(e1)` (Symmetry)
    - `e1.equals(e2) && e2.equals(e3) == e1.equals(e3)` (Transitivity)

Equality and hashCode()

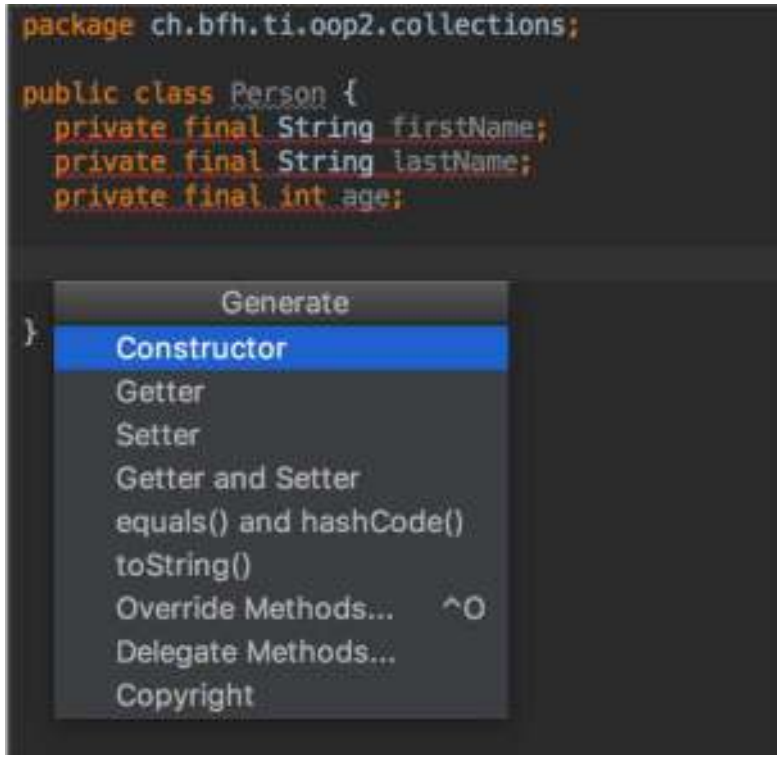
- Recommendation: When you override `equals(..)`, you should also override `hashCode()`.
- This allows you to use objects of your class as keys in hash tables (details later in course BTI7062 "Algorithmen und Datenstrukturen").

- The method `hashCode()` implemented in class `Object` only returns the memory address of the object in the JVM as an integer.
- If you need a `HashSet<MyClass>`, you **must** override `equals(Object obj)` and `hashCode()` in `MyClass`!

- Rules for `hashCode()`:
  - It two objects are equal as per `equals(..)` method, they must have the same hash code.
    → If `e1.equals(e2)` then `e1.hashCode() == e2.hashCode()`
  - Otherwise, `e1.hashCode() != e2.hashCode()` with very high probability.

Example: **Person** Class using IDE Automation from IntelliJ (1)

Generate Class and Constructor:



1. Define a `Person` class with its attributes `firstName`, `lastName`, and `age`.

2. Place cursor where you want IntelliJ to create the constructor.

3. Press ⌘N (Alt+Insert on Windows) to open "Generate" context Menu → Select "Constructor" → Click OK

Example: **Person** Class using IDE Automation from IntelliJ (2)

Generate Getters

```
package ch.bfh.ti.oop2.collections;

public class Person {
  private final String firstName;
  private final String lastName;
  private final int age;

  public Person(String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

}
```

Generate
- Constructor
- Getter
- Setter
- Getter and Setter
- equals() and hashCode()
- toString()
- Override Methods... ^O
- Delegate Methods...
- Copyright

1. Place cursor where you want IntelliJ to create the getters.

2. Press ⌘N (Alt+Insert on Windows) to open "Generate" context Menu → Select "Getter" → Select and mark "`firstName`", "`lastName`", and "`age`" → Click OK

Example: Person Class using IDE Automation from IntelliJ (3)

Generate `equals(..)` and `hashCode()`

```
package ch.bfh.ti.oop2.collections;

public class Person {
  private final String firstName;
  private final String lastName;
  private final int age;

  public Person(String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

  Generate
    Constructor
    Getter
    Setter
    Getter and Setter
    equals() and hashCode()
    toString()
    Override Methods...   ^O
    Delegate Methods...
    Copyright
}
```

1. Place cursor where you want IntelliJ to create the `equals(..)` and `hashCode()` methods.

2. Press ⌘N (Alt+Insert on Windows) to open "Generate" context Menu → Select "equals() and hashCode()" → Click Next 3x → Click Finish

Example: **Person** Class using IDE Automation from IntelliJ (4)

```java
import java.util.Objects;

public class Person {
  private final String firstName;
  private final String lastName;
  private final int age;

  public Person(String firstName, String lastName,
                int age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null || getClass() != o.getClass()) {
      return false;
    }
    Person person = (Person) o;
    return age == person.age &&
        Objects.equals(firstName, person.firstName) &&
        Objects.equals(lastName, person.lastName);
  }
  ...
```

Java Boilerplate Code generated by IntelliJ IDE.

```java
  ...
  @Override
  public int hashCode() {

    return Objects.hash(firstName, lastName, age);
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public int getAge() {
    return age;
  }
}
```

It is a sad state of affairs if you need the help of your IDE to be generate code for you just because your programming language is broken!

Complete code to create the same Person in Scala:

```
case class ScalaPerson(firstName: String, lastName: String, age: Int)
```

Complete code to create the same Person in Kotlin:

```
data class KotlinPerson(val firstName: String, val lastName: String, val age: Int)
```

The Kotlin and Scala compilers automatically generate the equivalent constructors, getters, **equals()**, **hashCode()**, and **toString()** for you.

See https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

HashSet<E>

- **HashSet<E>** uses a hash table internally to store the set.
  - It uses a **HashMap<E,Object>** with a dummy object as value and the set element as the key **E**.
- In a **HashSet<E>** the order of its elements is unspecified.
- **Example**

```
public class SetExample {
  public static void main(String[] args) {
    Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4));
    Set<Integer> s2 = new HashSet<>(Arrays.asList(3, 4, 5));
    Set<Integer> s3 = new HashSet<>(Arrays.asList(2, 5, 6));

    s2.addAll(s3);
    System.out.println(s2); // s2 = s2 union s3 = {2, 3, 4, 5, 6}

    s2.retainAll(s1);
    System.out.println(s2); // s2 = s2 intersect s1 = {2, 3, 4}

    s2.removeAll(s3);
    System.out.println(s2); // s2 = s2 minus s3 = {3, 4}

    s2.add(3);    // s2 = s2 union { 3 }
    s2.add(5);    // s2 = s2 union { 5 }
    s2.add(5);    // s2 = s2 union { 5 }
    System.out.println(s2); // s2 = {3, 4, 5}
  }
}
```

Hashable Objects should be Immutable

- Consider a **mutable** Person:
  - Instance variables are **not** final.
  - Class has public setter methods.

You get into trouble when you insert a mutable object into a hash set (or hash table).

Example:

→ **You no longer find the object!**

```
Person p1 = new Person("Alice", "Brown", 32);
Person p2 = new Person("Alex", "Jones", 24);
Person p3 = new Person("Bob", "Willson", 57);
Person p4 = new Person("Beth", "Smith", 50);

Set<Person> set = new HashSet<>(Arrays.asList(p1, p2, p3));

System.out.println(set.contains(p4)); // => false

System.out.println(set.contains(p3)); // => true

p3.setAge(58);  // modify instance variable => hash code changes
System.out.println(set.contains(p3)); // => false!
```

See https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

TreeSet<E>

- A **TreeSet** is not only a **SortedSet** but also a **NavigableSet**.
- **TreeSets** are implemented using a **TreeMap** underneath, which in turn is implemented as Red-Black Tree (see BTI7062 "Algorithmen und Datenstrukturen")

- The order in a **TreeSet<E>** is determined by
  - the natural order of **E** (**E** then must implement the **Comparable<E>** interface).
    Applied if set is constructed as **new TreeSet()** or **new TreeSet(otherCollection)**.
  - a **Comparator<E>** object **comp** that is provided in the constructor: **new TreeSet(comp)**.
- **Example**

```
List<Integer> list = Arrays.asList(4, 2, 5, 1);
NavigableSet<Integer> set1 = new TreeSet<>(list);
System.out.println(set1); // => [1, 2, 4, 5]  natural order of Integer

NavigableSet<Integer> set2 = new TreeSet<>(new Comparator<Integer>() {
  @Override
  public int compare(Integer o1, Integer o2) {
    return o2 - o1;
  }
});
set2.addAll(list);
System.out.println(set2); // => [5, 4, 2, 1]  order determined by Comparator object
```

Since **Comparator** is a functional interfaces the anonymous class can be replaced by the simple lambda: `(o1, o2) -> o2 - o1)`.

TreeSet<E>: Additional Examples

```
Person p = new Person("Bob", "Smith", 55);
NavigableSet<Person> set = new TreeSet<>(Arrays.asList(p));
```

ClassCastException: Person cannot be cast to Comparable. Class Person does not implement Comparable!

```
NavigableSet<Integer> s = new TreeSet<>(Arrays.asList(1, 3, 7, 2, 5, 4, 7));
System.out.println(s.headSet(5));   // [1, 2, 3, 4]  elements < 5
System.out.println(s.tailSet(5));   // [5, 7]  elements >= 5
System.out.println(s.subSet(2, 5)); // [2, 3, 4] elements 2..4

// Methods from NavigableSet<E>
System.out.println(s.floor(6));   // 5 (greatest element smaller or equal 6)
System.out.println(s.ceiling(6)); // 7 (smallest element greater or equal than 6)
```

HashSet<E> vs. TreeSet<E>

- **HashSet**
  - Elements are unordered.
  - Insert and Get are O(1) on average (= efficient, independent of size).

- **TreeSet**
  - Elements are ordered.
  - Insert and Get operations are O(log n) in the worst-case, n number of elements in **TreeSet** (= still considered rather efficient).
    - Note: Even though O(1) is better than O(log n) it does not necessarily mean that a **HashSet** is more efficient than a **TreeSet**, since BigO-Notation only refers to the *asymptotic complexity*! (details in BTI7065 "Berechenbarkeit und Komplexität")

- **Recommendation**: Use **HashSet** unless you need the elements stored in order.

Maps

HashMap, LinkedHashMap

TreeMap

Maps

- A `Map<K,V>` is an object that maps from key objects of type `K` to value objects of type `V`.
- Mathematically, a map is a function f: $K \rightarrow V$, that maps elements from domain K into domain V.

- **Example:**
  - Phone Book `Map<Person, PhoneNumber>`: A map from `Person` to `PhoneNumber`
  - Dictionary `Map<String, String>`: A map from German to English words.
  - Domain Name System `Map<Name, IPAddress>`: A map of names to IP addresses

- A map cannot contain duplicate key since each key can map to at most one value.
- Different keys are allowed to map to the same value (a map is an injective function).

- Maps are a generalization of arrays.
  - Array: access to elements through an integer index.
  - Map: access to values through a key object (e.g., `Person`, `String`, `Integer`)
- For this reason, maps are sometimes also called associative arrays.

Map<K,V> Interface

See https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Note: `Map<K, V>` does not extend the `Collection<E>` interface. It is the root of its own type hierarchy.

Methods	Description
`boolean containsKey(Object k)`	Returns `true` if the map contains a mapping with the given key.
`boolean containsValue(Object v)`	Returns `true` if the map contains a mapping with the given value.
`V get(Object key)`	Returns the value of the mapping for the given key. Returns `null` if the key does not exist.
`V getOrDefault(Object k, V defaultValue)`	Returns the value of the mapping for the given key if it exists. If the key does not exists, it returns the specified default value.
`V put(K key, V value)`	Adds a new mapping or replaces the value of an existing mapping. Returns the previous value of the mapping or `null` if there was no such mapping.
`V remove(Object key)`	Removes the mapping for the provided key, if such a mapping exists. Returns the previous value of the mapping or `null` if there was no such mapping.
`V replace(K key, V value)`	Replaces the value of a mapping for the key, but only if such a mapping already exists. Returns the previous value of the mapping or `null` if there was no such mapping.
`Collection<V> values()`	Returns a collection of all values.
`Set<K> keySet()`	Returns a set of all keys.
`int size()`	Return the number of key/value mappings in this map.

See https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

HashMap<K,V>

- In a HashMap the order of the keys is unspecified.
- The HashMap is implemented as a Hash Table.
  Uses chaining hash tables. Creates new hash table of double the size if load factor exceeds ¾. If the node list of a hash bucket grows beyond 8 elements, the linked list will be converted into a Red-Black tree for this bucket. Details in BTI7062 "Algorithmen und Datenstrukturen".
- Like the types E in a HashSet<E>, the key type K in HashMap<K, V> must be hashable.

Example:

```
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2);
map.put("three", 4);
map.put("three", 3);    // replaces the previous value

System.out.println(map);  // prints {one=1, two=2, three=3}
System.out.println(map.keySet()); // prints [one, two, three]
System.out.println(map.values()); // prints [1, 2, 3]

Integer i1 = map.get("one");
if (i1 != null) System.out.println(i1);    // prints 1

Integer i2 = map.get("four");
if (i2 != null) System.out.println(i2);    // no output

String key = "one";
if (map.containsKey(key))
  System.out.println(map.get(key)); // prints 1
```

This code is less efficient since it requires two look-ups into the hash table.

First lookup

Second lookup

See https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

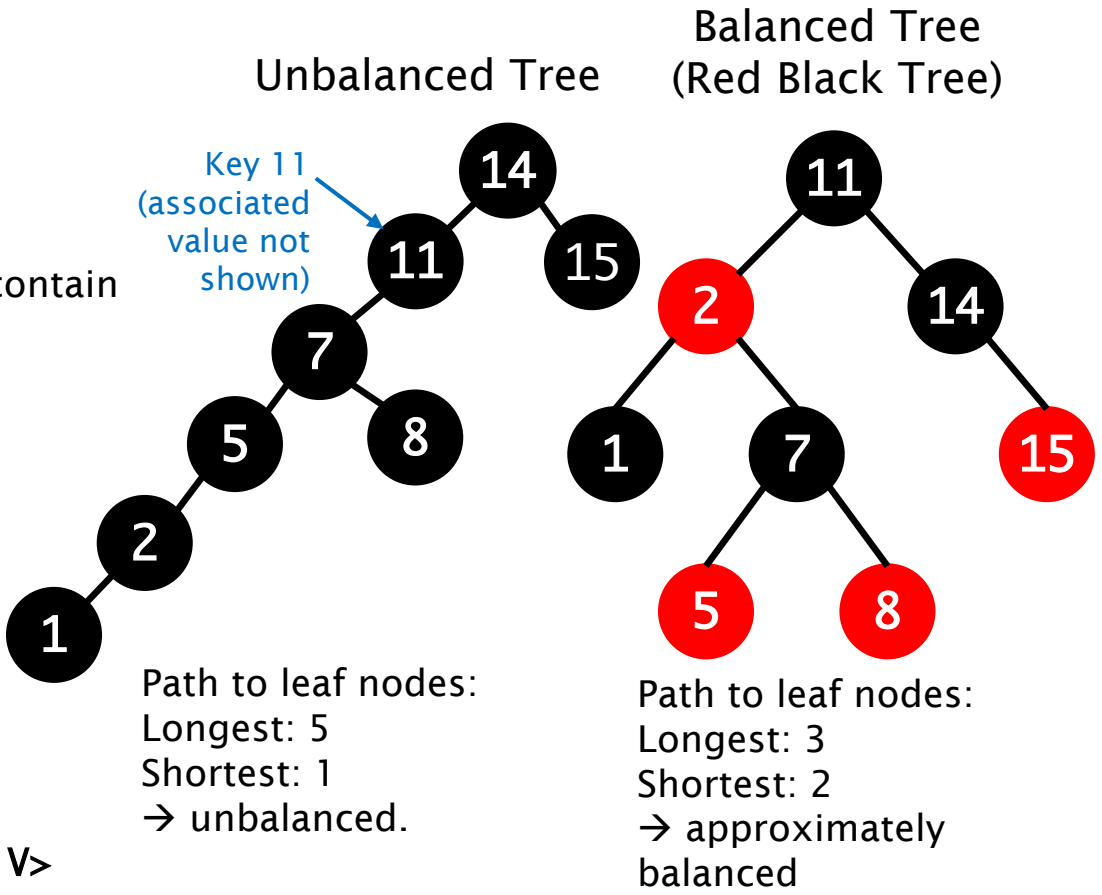
TreeMap<K,V>

- Based on Red-Black Trees
  (details in BTI7062 "Algorithmen und Datenstrukturen")
  - Red-Black Trees are self-balancing binary search trees.
  - A node can be either black or red
  - Every path from a node to any of its descendant leaf nodes must contain the same number of black nodes. → Balancing Property.
  - If a node is red, both of its children must be black.

`TreeMap<K, V>`

- The order of the keys is determined by
  - the natural order of K
    `new TreeMap<>()`: K must implement `Comparable<K>`.
  - a `Comparator<K>` object `comp` supplied to the constructor.
    `new TreeMap<>(comp)`

- Provides additional methods from the interfaces `SortedMap<K, V>` and `NavigableMap<K, V>`.
  - `Map.Entry<K,V> floorEntry(K k)`: greatest entry ≤ k
  - `Map.Entry<K,V> ceilingEntry(K k)`: smallest entry ≥ k
  - `Map.Entry<K,V> lowerEntry(K k)`: greatest entry < k
  - `Map.Entry<K,V> higherEntry(K k)`: smallest entry > k



Queues, Priority Queues, and Deques

ArrayDeque, LinkedList, PriorityQueue

See https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

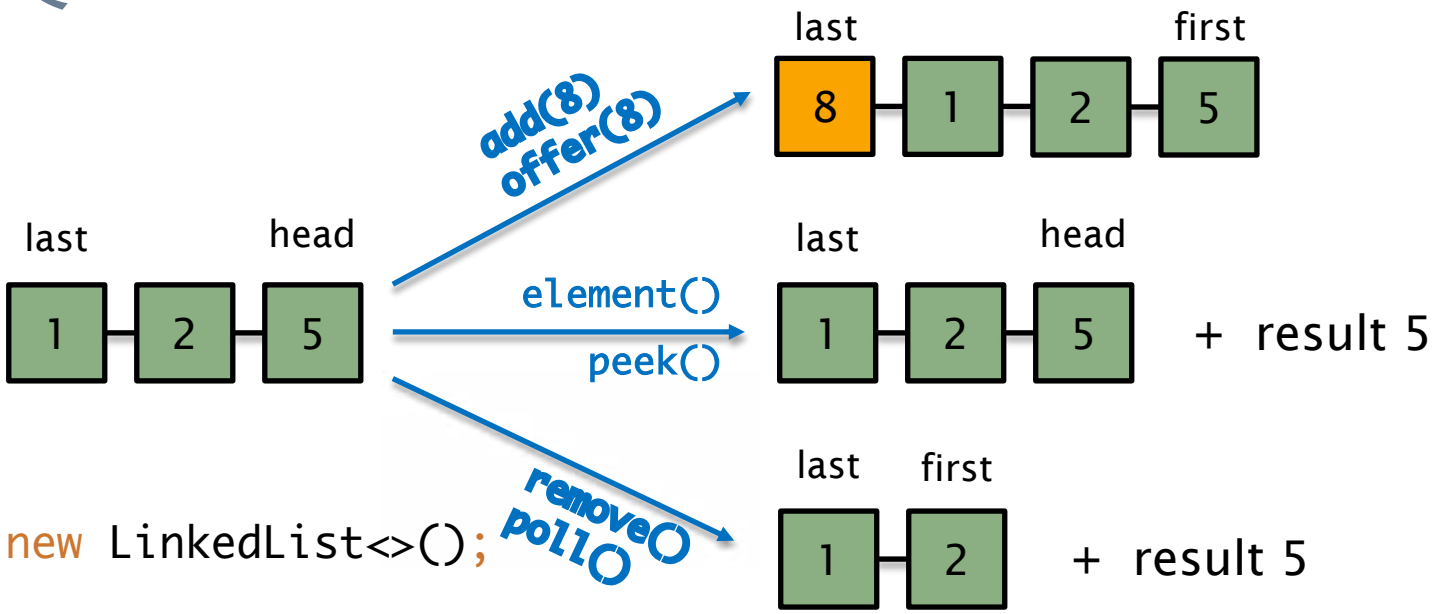
Queue<E> Interface

The classes `LinkedList`, `PriorityQueue`, and `ArrayDeque` implement this interface.

Additional Methods of Queue	Description
`boolean add(E e)`	Insert element to tail of the queue (returns `true`, throws exception if no space available).
`boolean offer(E e)`	Insert element to tail of the queue (returns `true` if element was added, `false` if no space available).
`E element()`	Returns the element at the head of the queue without removing it. Throws exception if queue is empty.
`E peek()`	Returns the element at the head of the queue without removing it. Returns `null` if the queue is empty.
`E remove()`	Removes and returns the head element of the queue. Throws an exception if the queue is empty.
`E poll()`	Removes and returns the head element of the queue. Returns `null` if the queue is empty.

The implementations in `LinkedList`, `PriorityQueue`, and `ArrayDeque` have all an unbounded capacity.

LinkedList as a Queue




```
Queue<Integer> queue = new LinkedList<>();

queue.add(5);
queue.add(2);
queue.add(1);
System.out.println(queue.element()); // => 5 (don't remove it)
System.out.println(queue.remove());  // => 5
System.out.println(queue.remove());  // => 2
System.out.println(queue.remove());  // => 1

queue.add(0);
System.out.println(queue.remove());  // => 0
System.out.println(queue.poll());    // => null
System.out.println(queue.remove());  // => NoSuchElementException
```

See https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

PriorityQueue<E>

- In a priority queues the entries are sorted by descending priority not by insertion order (like in normal queues).

- The order of in a `PriorityQueue<E>` is determined by
  - the natural order of the element type `E` as determined by the `Comparable<E>` it must implement.
  - a `Comparator<E>` object that is supplied to the constructor of `PriorityQueue`.

- The element are sorted in with ascending order.
  - Head element: smallest element in queue, smallest element → highest priority
  - Last element: largest element in queue, largest element → lowest priority.

See https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

LinkedList<E> Queue vs. PriorityQueue<E>

```
Queue<String> queue = new LinkedList<>();
queue.add("Eve");
queue.add("Bob");
queue.add("Zoe");
queue.add("Ellen");
while (!queue.isEmpty()) {
  System.out.println(queue.remove()); // => prints "Eve" "Bob" "Zoe" "Ellen"
}
```

Natural Ordering of String

```
PriorityQueue<String> prioQueue = new PriorityQueue<>();
prioQueue.add("Eve");
prioQueue.add("Bob");
prioQueue.add("Zoe");
prioQueue.add("Ellen");
while (!prioQueue.isEmpty()) {
  System.out.println(prioQueue.remove()); // => "Bob" "Ellen" "Eve" "Zoe"
}
```

Example PriorityQueue<E> with Comparator

```
class Employee {
  String name;
  int evaluationScore;          // result of annual performance review
  int yearsSinceLastPromotion; // number of years since last promotion
  public Employee(String name, int evaluationScore, int yearsSinceLastPromotion) { ... }
}
...
PriorityQueue<Employee> promotionQueue = new PriorityQueue<>(new Comparator<Employee>() {
  @Override
  public int compare(Employee o1, Employee o2) {
    return (o1.evaluationScore != o2.evaluationScore) ?
        (o2.evaluationScore - o1.evaluationScore) :
        (o2.yearsSinceLastPromotion - o1.yearsSinceLastPromotion);
  }
});

promotionQueue.add(new Employee("Bob", 4, 2));
promotionQueue.add(new Employee("Eve", 5, 3));
promotionQueue.add(new Employee("Zoe", 4, 5));
promotionQueue.add(new Employee("Joe", 2, 7));

int numPromotionsThisYear = 2;
for (int i = 0; i < numPromotionsThisYear && !promotionQueue.isEmpty(); i++) {
  System.out.println(promotionQueue.remove().name + " gets a promotion.");
}
```

Prioritization:
First by evaluation score,
then by years since last promotion.

Output:

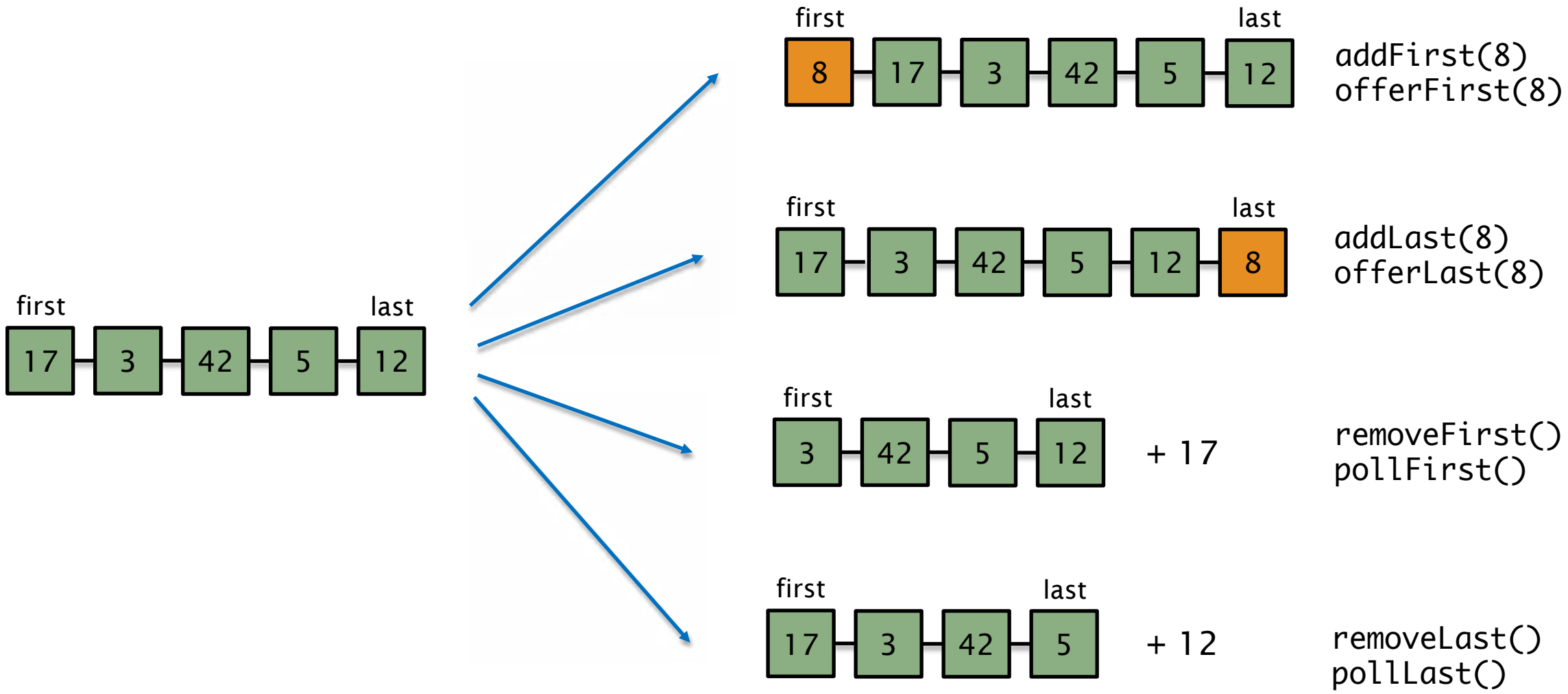
```
Eve gets a promotion.
Zoe gets a promotion.
```

See https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

Deque<E> Interface

Methods	Description
`E getFirst()`	Returns first element without removing it. May throw exception.
`E peekFirst()`	Returns first element without removing it. Returns `null` if empty.
`E getLast()`	Returns last element without removing it. May throw exception.
`E peekLast()`	Returns last element without removing it. Returns `null` if empty.
`void addFirst(E e)`	Insert element to front. Throws exception of deque full.
`boolean offerFirst(E e)`	Insert element to front. Returns `true` if there was space.
`void addLast(E e)`	Insert element to end. Throws exception of deque full.
`boolean offerLast(E e)`	Insert element to end. Returns `true` if there was space.
`E removeFirst(E e)`	Retrieves and removes first element. Throws exception if empty
`E pollFirst(E e)`	Retrieves and removes first element. Returns `null` if empty.
`E removeLast(E e)`	Retrieves and removes last element. Throws exception if empty
`E pollLast(E e)`	Retrieves and removes last element. Returns `null` if empty.

Deque<E> Interface – Illustration



LinkedList as a Deque

The **LinkedList<E>** class also implements the **Deque<E>** Interface.
ArrayDeque<E> also implements **Deque<E>** Interface but uses an array as the underlying representation.

```
Deque<Integer> deque = new LinkedList<>();

deque.addFirst(1);
deque.addFirst(2);
deque.addLast(3);
deque.addLast(4);
System.out.println(deque);  // prints [2, 1, 3, 4]

System.out.println(deque.removeFirst());  // prints 2
System.out.println(deque.removeLast());   // prints 4
System.out.println(deque.removeFirst());  // prints 1
System.out.println(deque.pollLast());     // prints 3
System.out.println(deque.pollFirst());    // prints null
System.out.println(deque.removeLast());   // throws NoSuchElementException
```

Summary: Java Collections Library

- The Java Collection Library contain a number of generic interfaces and classes to work with collections of objects.
- A collection is iterable, i.e., it can produce an iterator object that iterates over all elements of the collection.

- A `Collection<E>` is the root of the collection types, (except for Map)
- A `List<E>` is a collection with an order. Implementations:
  - `ArrayList<E>`: backed by arrays
  - `LinkedList<E>`: backed by doubly linked list.
- `Queue<E>`: is a list with FIFO (first-in first-out) access
- `Stack<E>`: is a LIFO (last-in first-out) queue
- `Deque<E>`: is a Double-ended Queue, allowing appends and removals on both ends.
- `PriorityQueue<E>`: is a queue that retrieves the element based on the objects' priorities (using `Comparable<T>` or `Comparator` object and in ascending order).

Sets

- `HashSet<E>` ← use this unless you need ordering
  - Elements are unordered.
  - The element type must override `equals(..)` and `hashCode()`.
- `TreeSet<E>`
  - Elements are ordered.
  - Element type must implement `Comparable`. interface or you need to provide a `Comparator` object to the constructor of the `TreeSet`.

Maps

- `HashMap<K,V>` ← use this unless you need ordering
  - Entries are unordered, i.e., not ordered by key.
  - The key type must override `equals(..)` and `hashCode()`.
- `TreeMap<K,V>`
  - Entries are ordered by key.
  - Key type must implement `Comparable`. interface or you need to provide a `Comparator` object to the constructor of the `TreeMap`.

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

Multithreading

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS 2018

Book:

- Chapter 14: "Concurrency" in Cay S. Horstmann, "Core Java", 10th Edition.
- Brian Goetz et al., "Java Concurrency in Practice", Addison-Wesley, 2006.

Outline Topic 9: Multithreading

- Threads
- Race Conditions
- Mutual Exclusion
- Deadlocks
- Synchronization of Threads
- Callables and Futures
- Synchronized Collections

Motivation for Parallel Programs

Moore's Law, Dennard Scaling

Moore's Law

- Prediction of Gordon Moore (Intel co-founder) in 1965:
  - The number of components that can be put on a chip economically doubles every year.
- Moore revised rate in 1975:
  - Doubling every two years.
- Common misconception:
  - Moore's Law = Performance doubles every 18 months.
  - Moore, in its original form, only makes statement about doubling of economical integration density.
  - Performance doubling every 18 months includes other factors too:
    - Dennard Scaling
    - Improvements in Microarchitecture

y-axis is logarithmic.

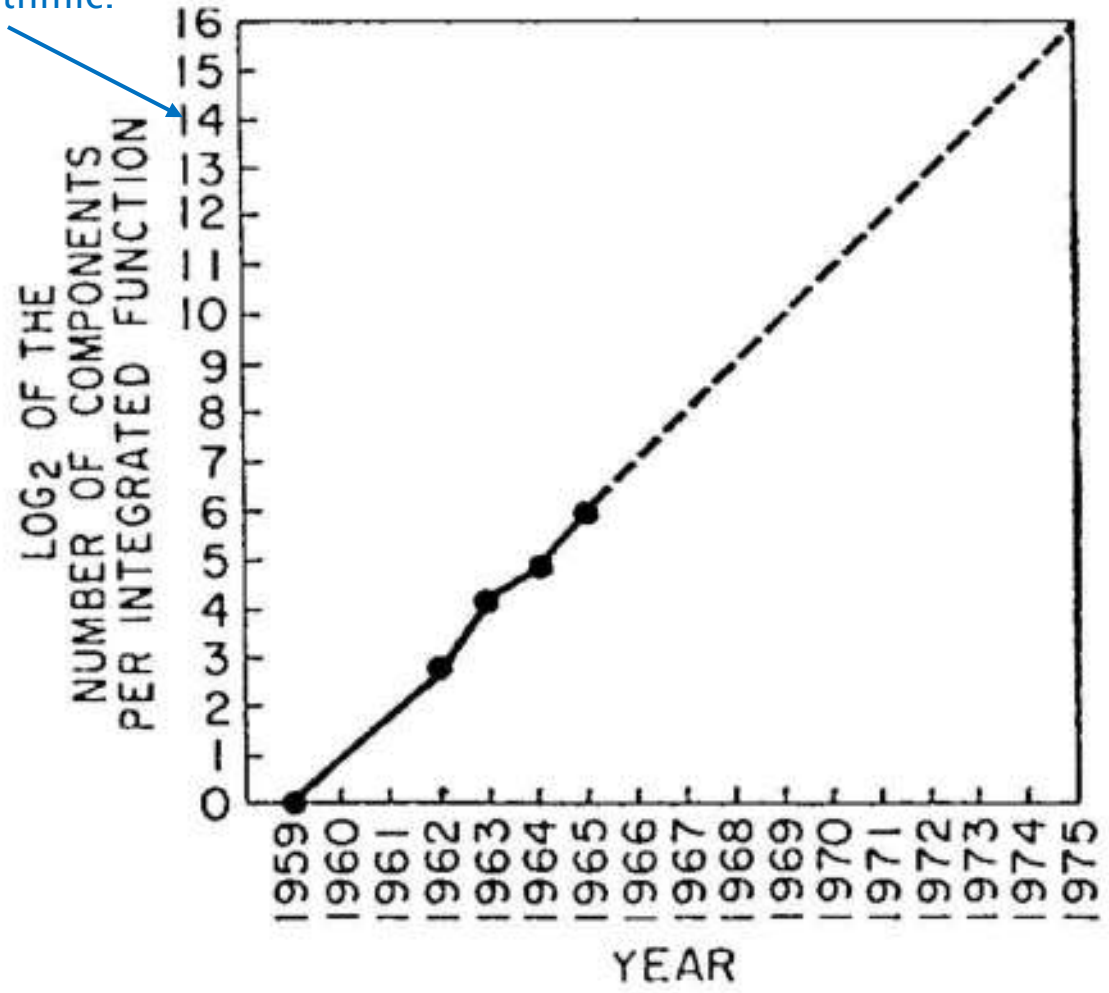
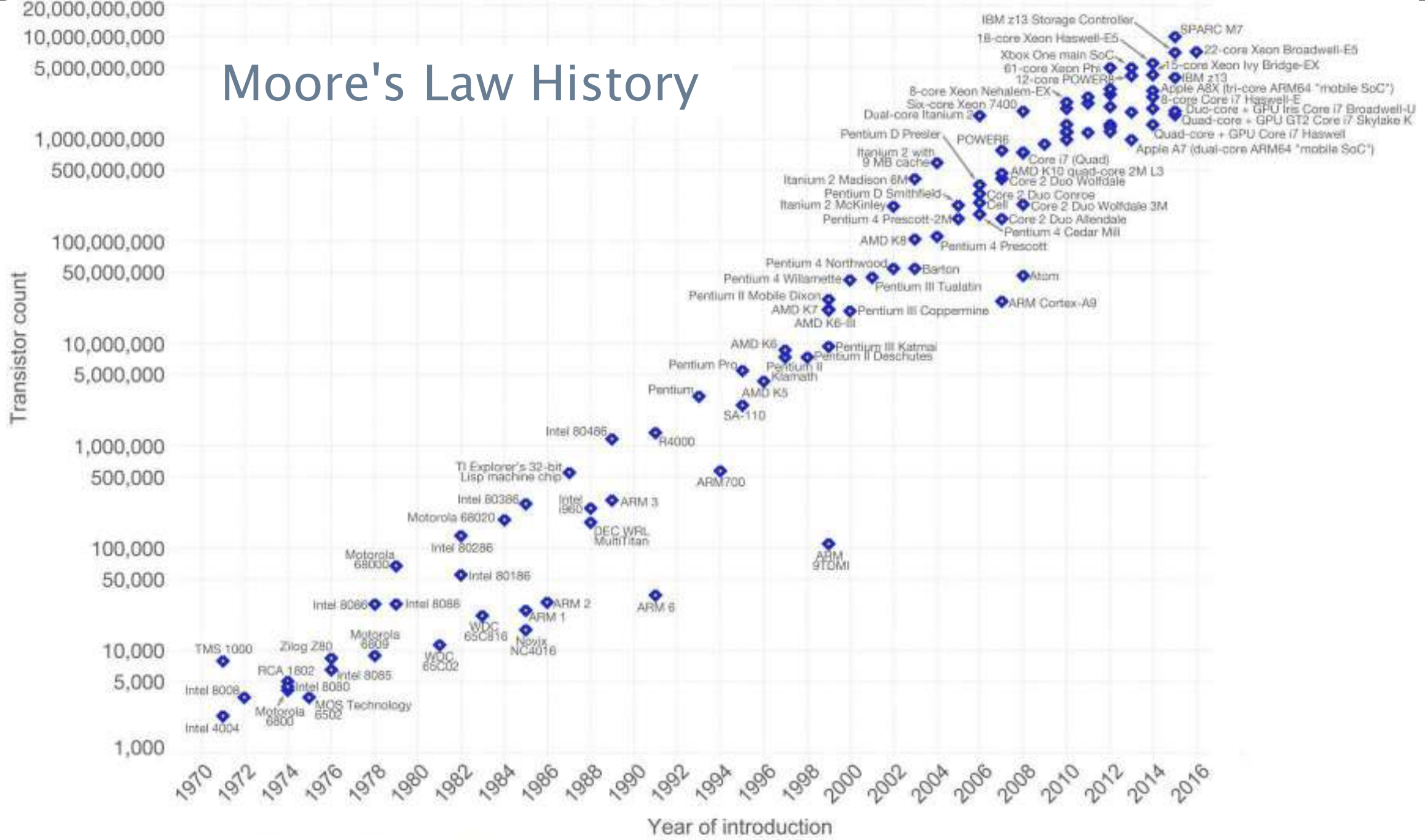


Figure from G. Moore's 1965 Paper "*Cramming more Components into Integrated Circuits*".
(Moore was quite bold to extrapolate from 5 points, i.e., to connect 5 points to a straight line!)

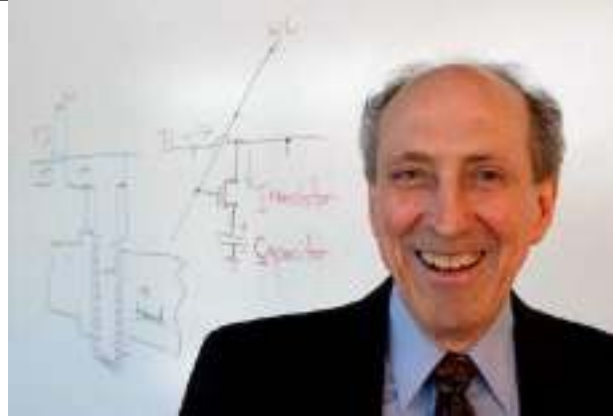
Moore's Law History



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

Dennard Scaling

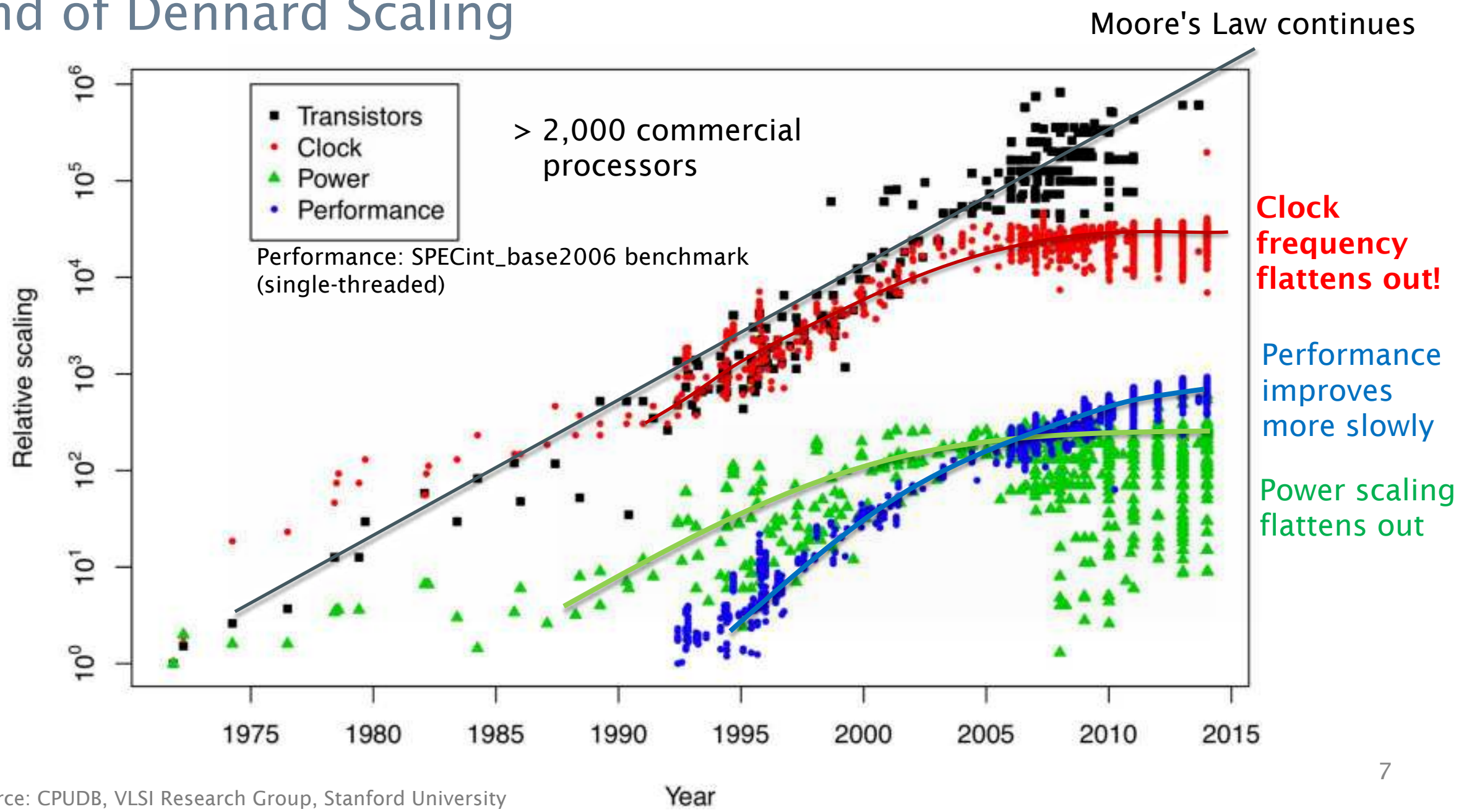


Robert Dennard, 1932
Inventor of the DRAM

- Prediction by Bob Dennard (IBM, 1975)
  - As transistors get smaller, the power density (W/cm^2) stays constant.
  - Current and voltage are proportional to feature size (e.g., 130 nm, 90 nm, 65 nm)
- The smaller the features (transistors) the faster a chip can operate.
  → Higher clock rates (GHz)
- Unfortunately, Dennard's Scaling Law ignores "leakage current" and the "near-threshold voltage regime" when reducing transistor dimensions into the sub-micron range.

  → Power density now increases as transistors get smaller!
  → Dennard Scaling ended around 2006.

End of Dennard Scaling



Data Source: CPUDB, VLSI Research Group, Stanford University

Rise of Multi-Core Processors

- Moore's Law continues → we can still have more transistors/chip
- But we cannot increase the clock frequency as we make transistors smaller.
  - Program do not simply run faster.
  - We can still make processors faster by microarchitectural improvements, but not as aggressively anymore (out-of-order processing, vector units, larger caches).

- What do with the additional transistors?

- Put more processor cores on a chip.
- However, in order to take advantage of these cores, the programs have to be written with parallelism in mind!

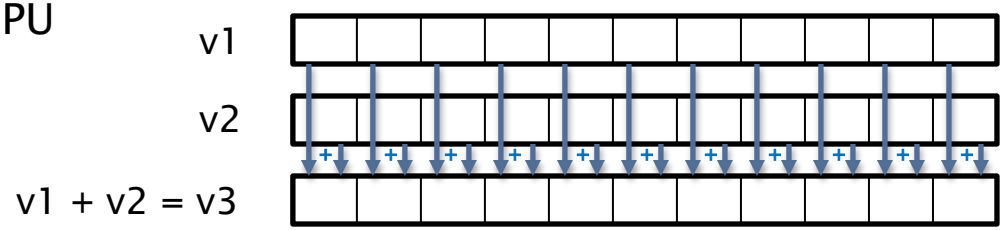
Different Types of Parallelism

Task Parallelism

- Each processor executes a different program, process, or operation on different data.
- **Example**: CPU1 runs physics simulation, CPU2 game AI, and CPU3 sound generation in a computer game.

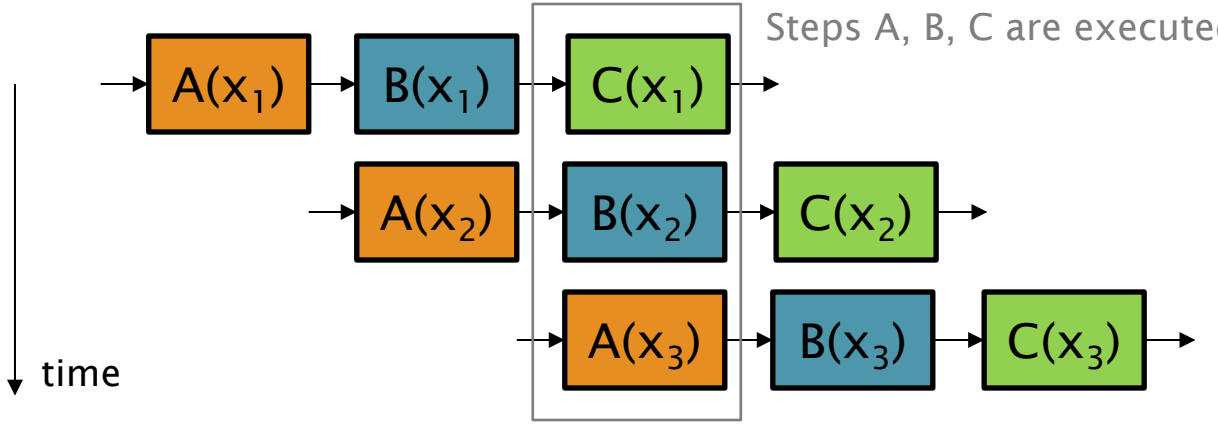
Data Parallelism

- Processor applies the same operation on multiple data items at the same time.
- **Example**: Addition of two vectors in a vector unit of a CPU



Pipeline Parallelism

- Execution of split steps A → B → C that can be executed concurrently on three units.



Steps A, B, C are executed at the same time but on different data items.

Example: Laundry
A: Washing Machine
B: Dryer
C: Iron

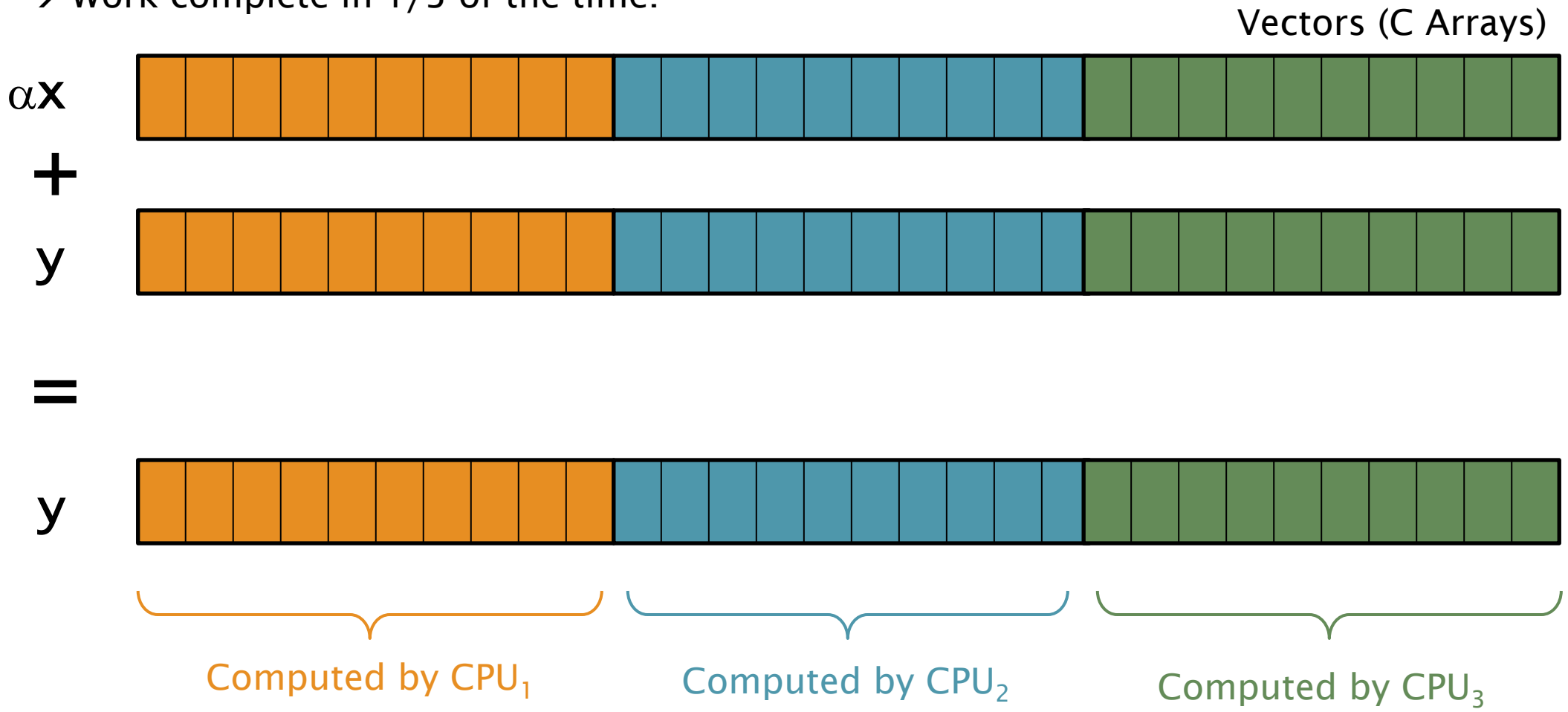
Writing Parallel Programs – Example Vector Addition

- Consider this non-parallel, i.e., sequential, implementation of vector addition in C.
- Adds a vector to another vector weighted by a scalar α: $\vec{y} := \vec{y} + \alpha\vec{x}$
  (Function is called daxpy "double precision ax plus y" in the BLAS library)

```
void daxpy(int n, const double alpha, const double * const x, double * const y)
{
  for (int i = 0; i < n; i++)
  {
    y[i] = y[i] + alpha * x[i];
  }
}
```

Parallelizing Vector Addition

- **Idea**: Slice vectors and give each processor a slice to work on independently.
- **Example**: Work is split among 3 CPUs → Each CPU does 1/3 of the work
  → Work complete in 1/3 of the time.



Parallel Vector Addition in C using OpenMP Annotation

- OpenMP are compiler directives that instruct the compiler to automatically generate parallel code for certain sections (basic blocks) in your program.
- A compiler that does not support OpenMP will simply ignore these directives and produce a traditional, non-parallel executable binary.

```
void daxpy(int n, const double alpha, const double *const x, double *const y)
{
  #pragma omp parallel for
  for (int i = 0; i < n; i++)
  {
    y[i] = y[i] + alpha * x[i];
  }
}
```

Instructs the compiler to split the for loop into independent pieces that are executed by a number of parallel processing threads (corresponding to number of logical CPU cores).

How Fast is Parallel Code?

- Performance Evaluation on MacBook Pro 15" (4 physical CPU cores, i7-7920HQ, Kaby Lake) and Clang 6.0 Compiler.
- Vector size 1,000,000 elements, average of 1,000 runs:

- Sequential Code: 972 μsec / vector addition (average of 1000 iterations)
- Parallel Code with OpenMP: 842 μsec / vector addition (average of 1000 iterations)

- Speedup from Parallelization = $\frac{972\ \mu sec}{842\ \mu sec} = 1.15$ → 15% faster.

- Parallel program uses all four cores, so work per core should be ¼.
  → Program should be 4× faster, but is only 1.15×! *What is wrong here?*

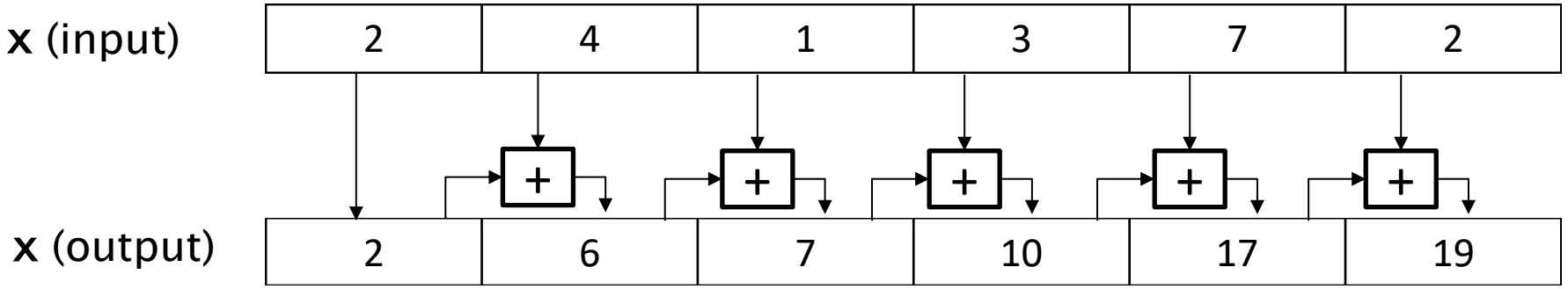
- Challenge 1: How to make parallel code run fast?

Challenge 2: How to Write Parallel Code?

- Consider this simple sequential program: Prefix sum in an array.

```
void prefix_sum(unsigned int n, int * const x)
{
  for (int i = 1; i < n; i++)
  {
    x[i] = x[i] + x[i-1];
  }
}
```

Value for current iteration i depends on value of previous iteration i-1.



- How can we parallelize this program?
  - There are data dependencies between successive loop iterations.
  - We cannot just sprinkle in some `#pragma omp` to have it automatically parallelized.

Challenge 3: How to Scale Parallel Code?

- The degree of parallelism may range from a few cores (mobile) to a few tens (enterprise servers) to thousands (GPUs).
- Writing a parallel program to run on 2 processors is very different from writing one for 2,500 processors.

Mobile

Google Pixel 2
(Snapdragon 835)

8 CPU cores in big.LITTLE architecture, 4 small and 4 big cores)

Enterprise

Lenovo X3650 m5
Two CPU Sockets

2x 22 CPU cores
= **44 CPU cores/system**

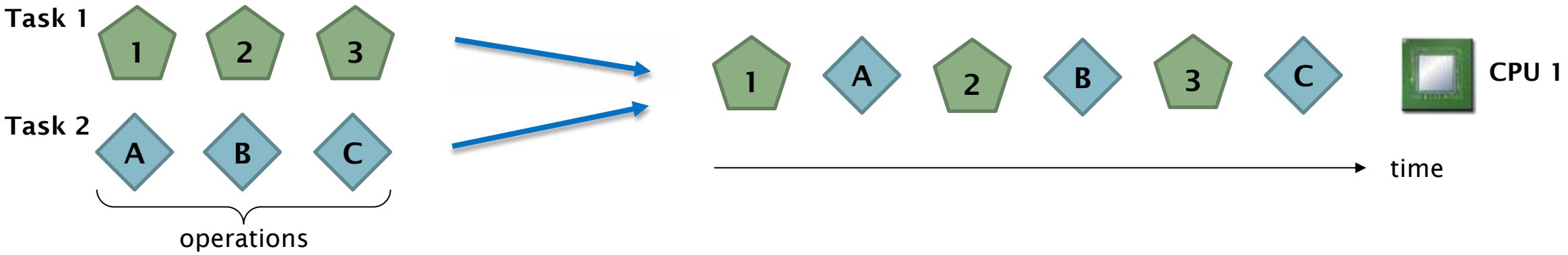
Artificial Intelligence, Compute, Gaming

NVIDIA GTX 1080 GPU
20 processors à 128 cores
= 2,560 cores! (*)

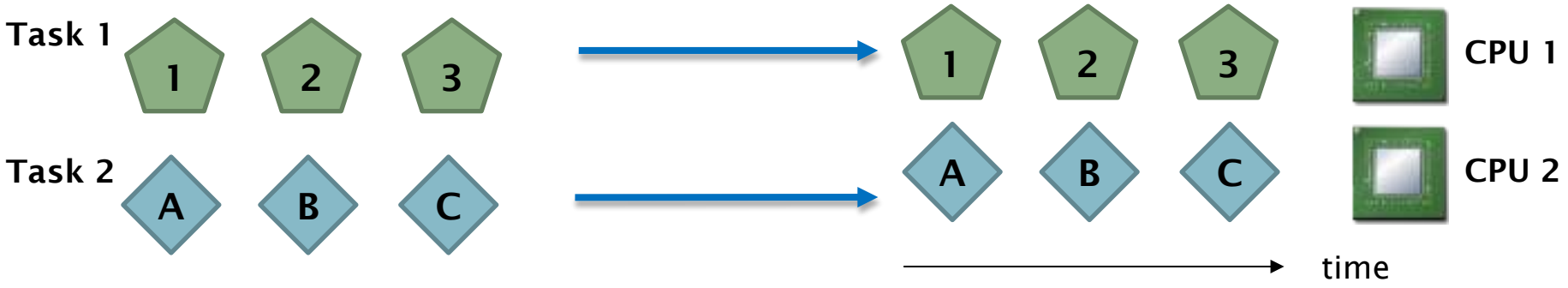
*) A core on a GPU is not a full CPU core. It is more like an ALU, but it nevertheless can run one complete thread.

Concurrency and Parallelism

- Concurrency: Composition of independently executing things.
  - It is about dealing with lots things at once.
  - Things run at the same time (execution overlaps) but do not necessarily run at the same time.
  - Example: Executing Word and Excel concurrently on a single-processor computer.



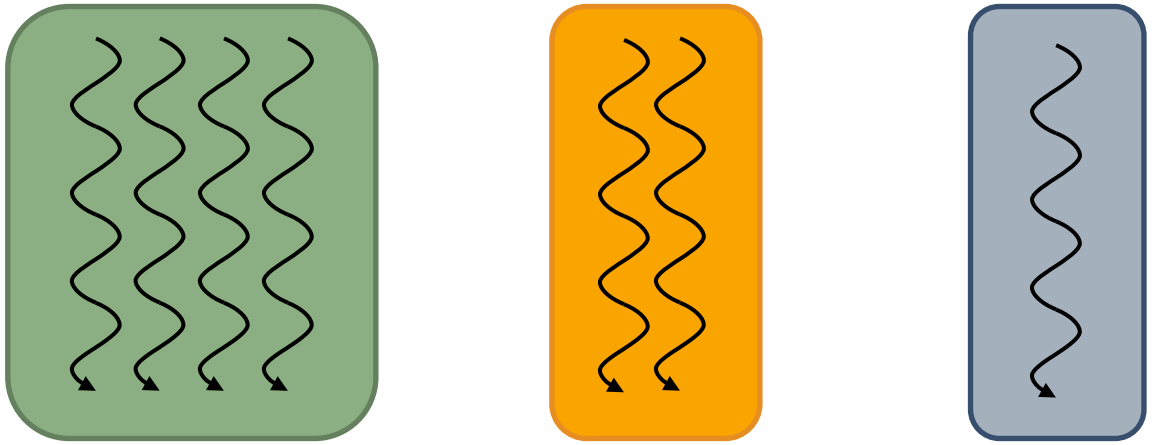
- Parallelism: Simultaneous execution of possibly related computations.
  - It is about doing lots of things at once.
  - Example: Watching a Movie while transcribing a video and downloading a big file.



Threads and Processes

- A thread is the sequence of instructions executed by a processor (today processor *core*).
- A process is the instance of a program that is being executed.
  - A process consists of at least one thread.
  - A multi-threaded program will use more than one thread when being executed.
- Threads within a process share the same memory space.
- Memory spaces of processes are isolated from each other.

A thread



Multithreaded Process A with 4 threads

Multithreaded Process B with 2 threads

Single-threaded Process C

CPU Cores

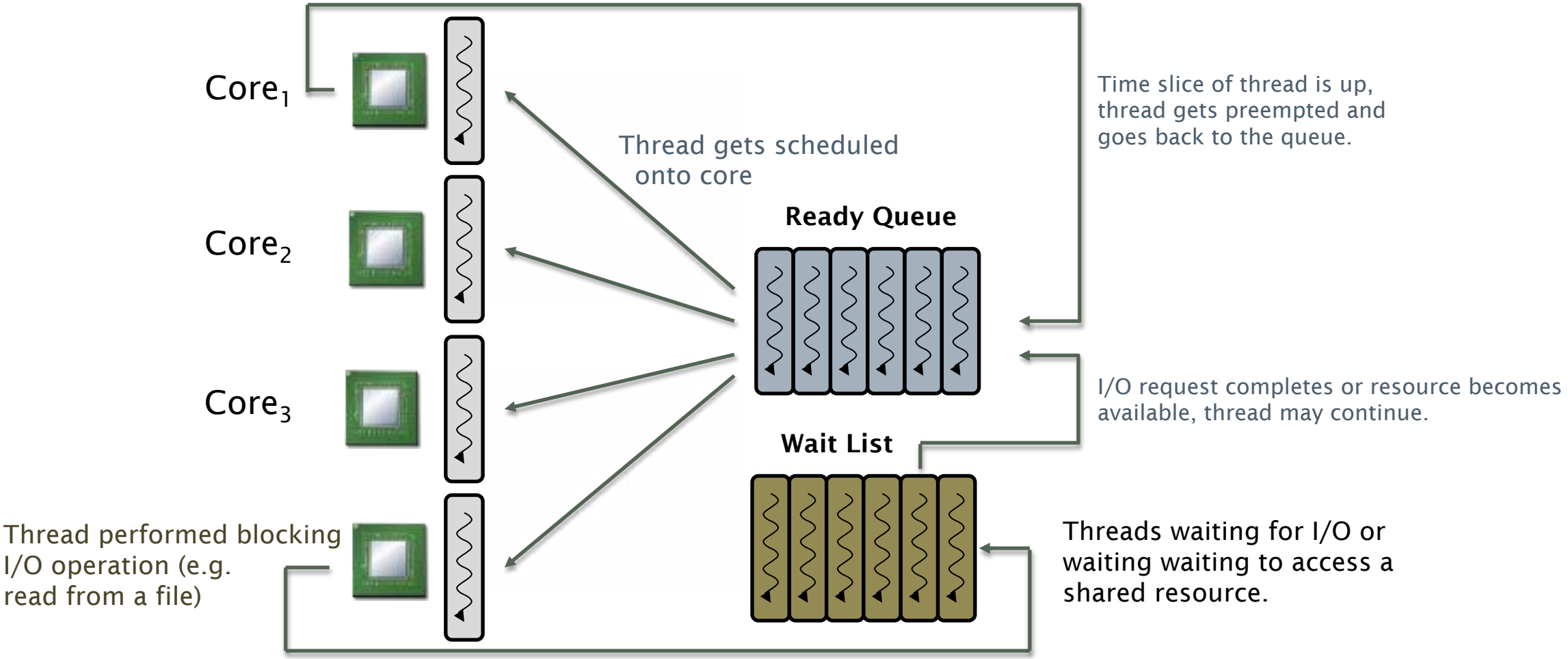
- CPU cores execute threads.
- Simultaneous Multithreading SMT
  - A physical CPU core can execute multiple threads concurrently
  - A physical CPU core appears to the OS as *n* logical cores.
  - Each logical core gets its own register file and instruction decoder but share execution units (integer, vector, and load/store units).
  - Intel Hyper Threading: n = 2, two logical cores/physical core
  - IBM POWER8: n = 8, 8 logical cores/physical core

Heterogenous Cores in
ARM big.LITTLE architecture

		CPU Sockets	Physical Cores	max. Concurrent Threads/System
Mobile	Google Pixel XL (Snapdragon 385)	1	4 small + 4 big	8
Laptop	MacBook Pro (Intel i7-7920HQ)	1	4	8
Server	Lenovo x3650 M5	2	2x 22 = 44	44x 2 = 88
Server	IBM Power E880C (max. config)	4 drawers x 4 SCMs = 16	16 x 12 = 192	192x 8 = 1,536
GPU	NVIDIA GTX1080 Ti	1	28x 128 = 3,584	28x 2048 = 57,344

Scheduler

- The scheduler assigns threads to CPU cores.
- There are usually more threads than CPU cores.
  → Each thread gets a CPU core for some short time (a time slice).
  → When time slice is up, OS takes CPU away and reassigns it to another thread (preemptive multithreading).



Concurrency on Single-Core Systems?

- Concurrent execution **can help** on systems with one single CPU

Example:

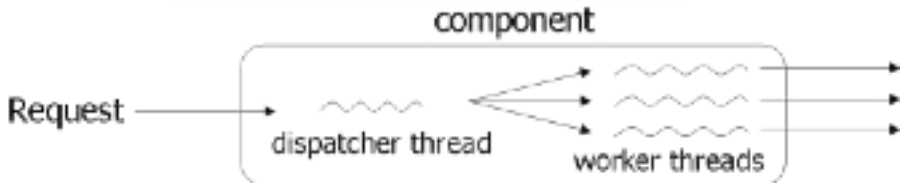
- A webserver (Apache) handling requests from browsers over a TCP/IP connection.

1. Client connects to Web Server.
2. Web Server assigns a thread that handles this connection.
3. Thread blocks waiting for the next HTTP request from the client over this connection. Once data is available on this connection the thread wakes up.
4. Thread handles the request.
5. Thread waits for the next input or until the connection is closed.

- Each client connection is handled by a separate thread → independent connections.
- Threads are waiting for data from the clients most of the time (#threads >> #cores).
- State of the connection can be captured in the thread. OS wakes them up if there is data and work to do.

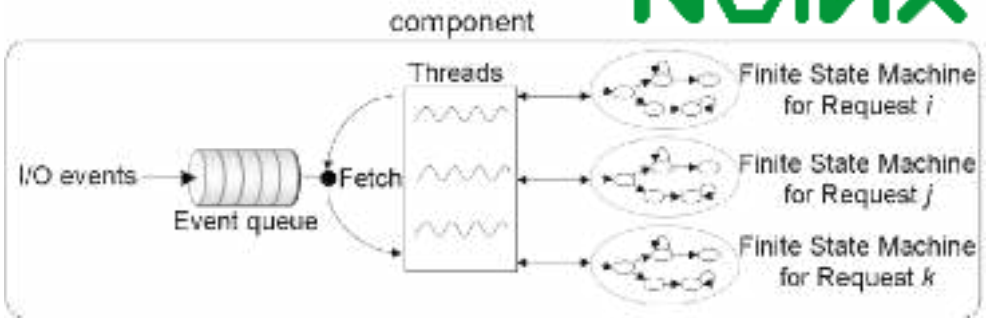
- But…
  - each thread requires resources (e.g., separate stack and state kept in operating system kernel) even if it is waiting and not using the CPU.
  - the number of threads is limited (a few 1,000 max)

Example Web Server: Apache vs. NGINX



- Apache Web server released 20+ years ago.
- Each new connection forked its own process (heavy-weight)
- Later on, thread pools and processes with multiple worker threads were added.
- However, a webserver serving 1000s of connections cannot have 1000s threads.

- NGINX is a web server that follows an event-driven approach. It uses a small number worker threads that handle many connections and their associated state.
  - Reactive approach: workers react on changes
  - Results in a state-machine based design
  - More difficult than a thread-based solution, however, if done right, more efficient.



- The IT community oscillates between threaded and event-driven approaches every 10 years. Currently, event-driven is more popular. In a few years, IT professionals will discover, *again*, that programming with state machines is more difficult than using threads.

Threads in Java

Working with Threads

Creating Threads in Java: Runnable Approach

- When Java program starts, the JVM uses one single thread, called main thread, to run the `main()` method.
- Additional threads can be created explicitly using `new Thread(…)`.
- Code of the additional threads is contained in the `void run()` method defined by the Interface `Runnable`.

```
public class YourRunnable implements Runnable {

  @Override
  public void run() {
    // Your thread code
  }
}
```

- The `Thread` constructor accepts a `Runnable`, that contains the code of the thread.
- The thread corresponding to the newly created `Thread` object must be started by calling the `start()` method on the `Thread` object.

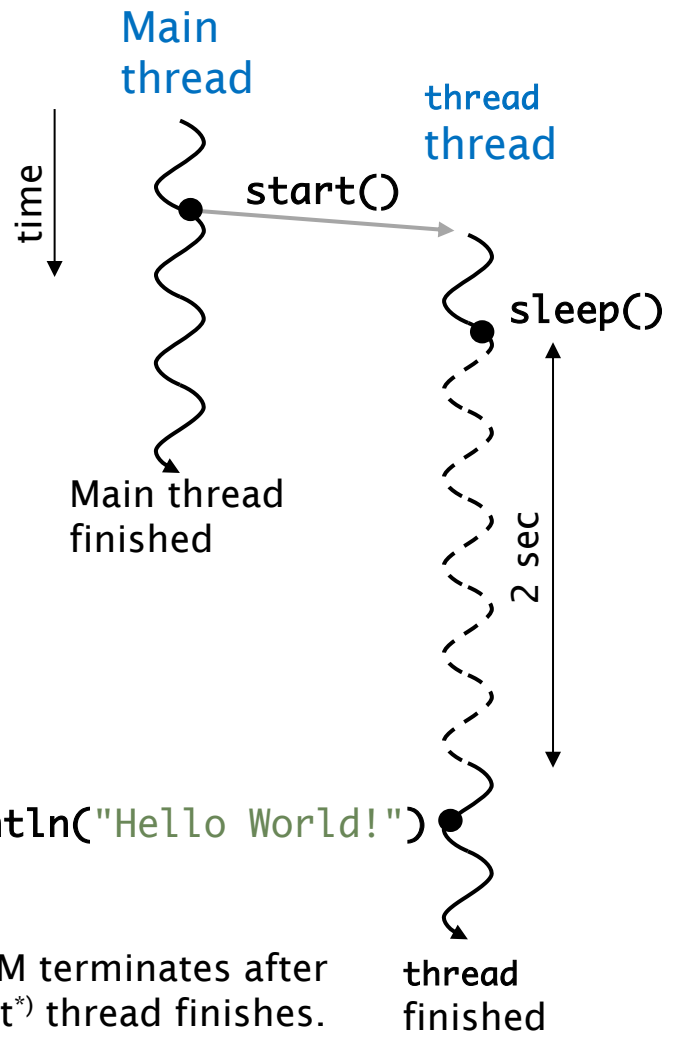
```
Thread thread = new Thread(new YourRunnable());
thread.start();
```

Example: Runnable Approach

```java
public class HelloWorld implements Runnable {

  @Override
  public void run() {
      try {
        Thread.sleep(2000); // Thread sleeps for 2 sec
      } catch(InterruptedException ignored) { }
      System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    Runnable helloWorld = new HelloWorld();
    Thread thread = new Thread(helloWorld);
    System.out.println("Starting thread...");
    thread.start();
    System.out.println("done");
  }
}
```

Output

```
Starting thread...
done
Hello World!
```



The JVM terminates after the last*) thread finishes.

*) More precisely, the JVM terminates after every *non-daemon* thread has finished.

Example: Runnable Approach with Anonymous Class

```
public class HelloWorldAnonymous {

  public static void main(String[] args) {
    Runnable helloWorld = new Runnable() {
      @Override
      public void run() {
        try {
          Thread.sleep(2000); // Thread sleeps for 2 sec
        } catch(InterruptedException ignored) { }
        System.out.println("Hello World!");
      }
    };

    Thread thread = new Thread(helloWorld);
    System.out.println("Starting thread...");
    thread.start();
    System.out.println("done");
  }
}
```

helloWorld points to an instance of an anonymous class that implements the Runnable Interface.

Example: Runnable Approach with Lambda (since Java 8)

- Runnable is a functional interface in Java 8 → It can thus be substituted by a lambda expression.

```
public class HelloWorldLambda {
  public static void main(String[] args) {
    Runnable helloWorld = () -> {
      try {
        Thread.sleep(2000); // Thread sleeps for 2 sec
      } catch(InterruptedException ignored) { }
      System.out.println("Hello World!");

    };

    Thread thread = new Thread(helloWorld);
    System.out.println("Starting thread...");
    thread.start();
    System.out.println("done");
  }
}
```

Threads by Extending Thread class (not recommended)

- The code of the thread can also be specified by extending the Thread class and overriding its `run()` method.
- The thread's own `run()` method is only called if the constructor is invoked without a `Runnable` object.
- Given Java's single-inheritance model, extending Thread "takes away" the possibility to extend from any other class. Better use `Runnable`.

```
public class HelloWorldExtension extends Thread {

  @Override
  public void run() {
    try {
      Thread.sleep(2000); // Thread sleeps for 2 sec
    } catch(InterruptedException ignored) { }
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    Thread thread = new HelloWorldExtension();
    System.out.println("Starting thread...");
    thread.start();
    System.out.println("done");
  }
}
```

Stopping Threads

- A thread terminates when its `run()` method finishes.

- Threads cannot be forcibly "killed". They need to be notified via a termination condition, which they check periodically and then terminate "voluntarily".
  - Do **not** use the now deprecated `stop()` method on thread! (another design flaw in Java)
  - Why is `stop()` a bad idea? You don't know what the thread was doing when it got killed.
  - It might have been in the middle of manipulating a linked listed → linked list would be in an inconsistent state afterwards!

- Use `interrupted()` Method to signal the thread it should terminate (or interrupt) its work. This method sets a special flag, called interrupt status flag, on the thread.
- Your thread code should periodically check this flag via `Thread.interrputed()` and terminate if it finds the flag set. Calling this method will clear the flag.
- Note you can choose to ignore this flag in your thread.
- A thread waiting in a `Thread.sleep(…)` will also woken up by receiving an `InterruptedException` and the flag is cleared too.

Example: interrupted()

```
public class AbortThread {
  public static void main(String[] args) {
    Thread pinger = new Thread(() -> {
      while (!Thread.interrupted()) {
        System.out.println("ping");
        try {
          Thread.sleep(1000);
        } catch(InterruptedException ignored) {
          System.out.println("interrupted");
          return;
        }
      }
    });
    pinger.start();

    // Main thread waits 5 sec before interrupting pinger thread
    try {
      Thread.sleep(5000);
    } catch(InterruptedException ignored) { }
    System.out.println("interrupting...");
    pinger.interrupt();
  }
}
```

Flag already cleared here so we need to return.

Output

```
ping
ping
ping
ping
ping
interrupting...
interrupted
```

Download file `pendulum.zip` from Moodle.

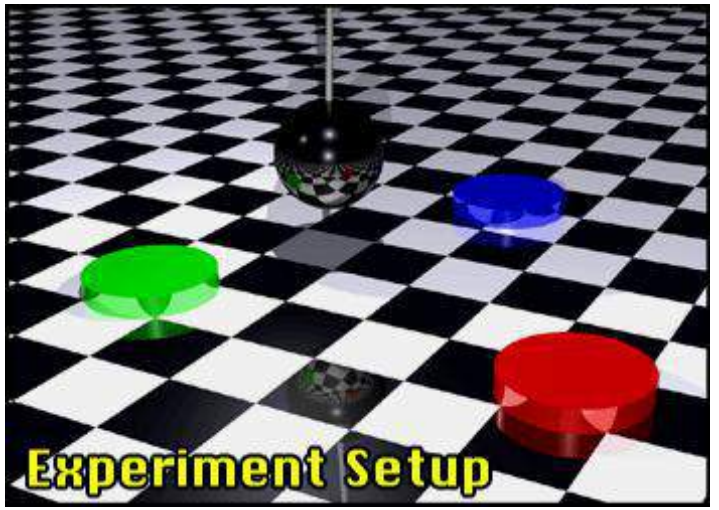
Use Case: Pendulum "Fractal"

Embarrassingly Parallel Code: Pixels can be computed independently and easily assigned to threads.

A Compute-heavy Example

Magnetic Pendulum Simulation

My almost 20-year old hobby project to test the performance of parallel computers.



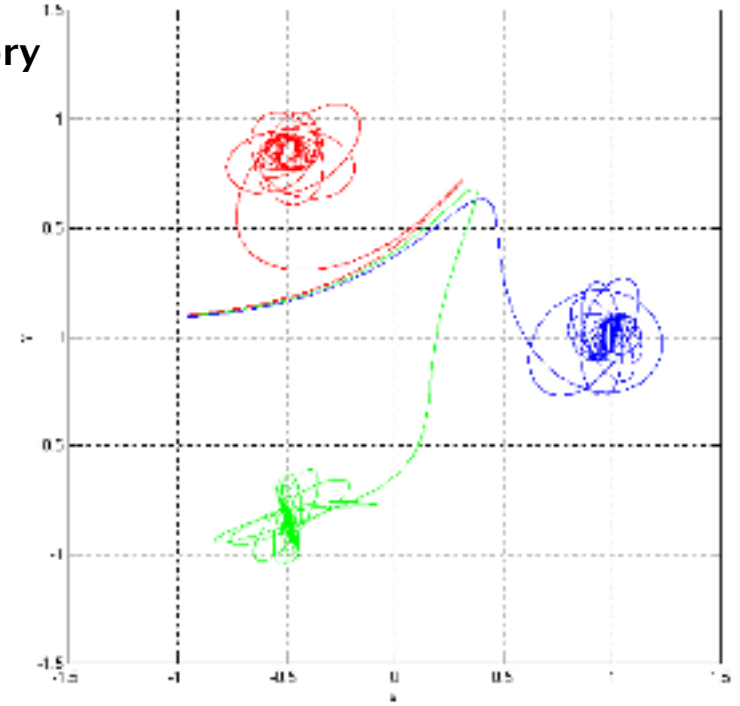
- Iron pendulum suspended over three magnets
- Simulate trajectory of pendulum.
- Figure out over which magnet the pendulum comes to rest (system is dissipative, it gradually loses energy)

- Movement of pendulum can be described by a coupled system of ordinary differential equations (ODEs).
- Here, in simplified form:

$$
\begin{aligned}
\ddot{x}(t) + R\dot{x}(t) + Cx(t) &= \sum_{i=1}^{3} \frac{x_i - x(t)}{\left(\sqrt{\left(x_i - x(t)\right)^2} + \left(y_i - y(t)\right)^2 + d^2\right)^3} \\
\ddot{y}(t) + R\dot{y}(t) + Cy(t) &= \sum_{i=1}^{3} \frac{y_i - y(t)}{\left(\sqrt{\left(y_i - x(t)\right)^2} + \left(y_i - y(t)\right)^2 + d^2\right)^3}
\end{aligned}
$$

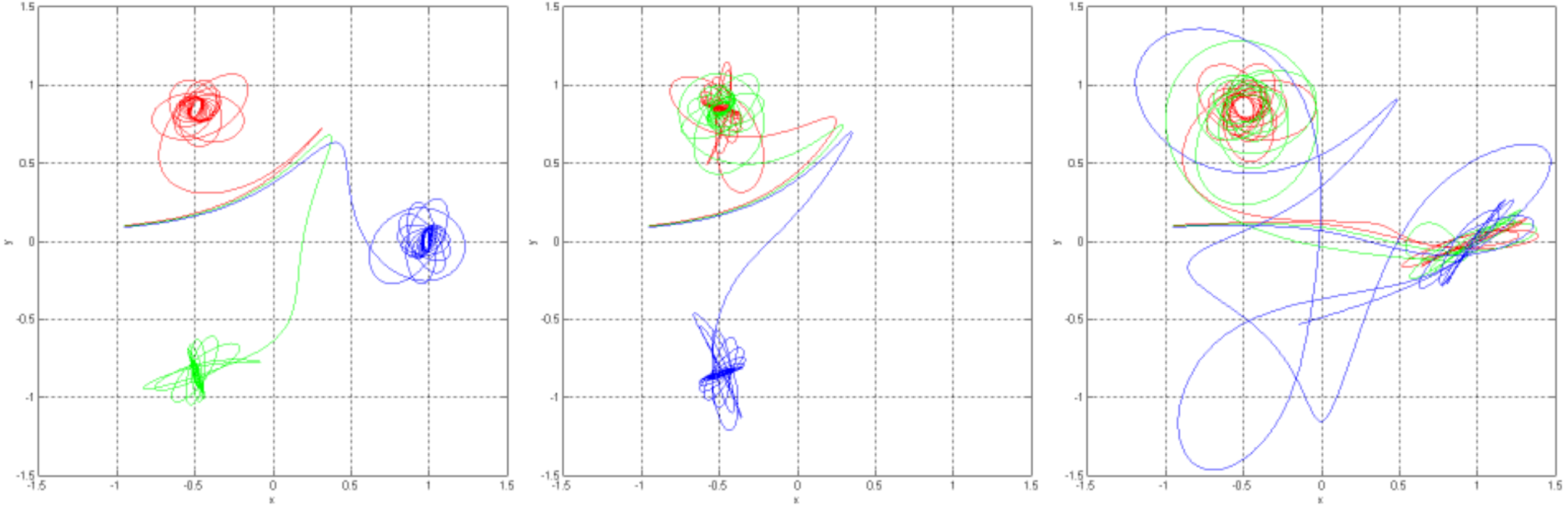
- Trajectories are computed by numerical integration (Beeman).

Trajectory



Final Position is Extremely Sensitive to the Start Position

Almost identical start position but completely different trajectories and end positions.

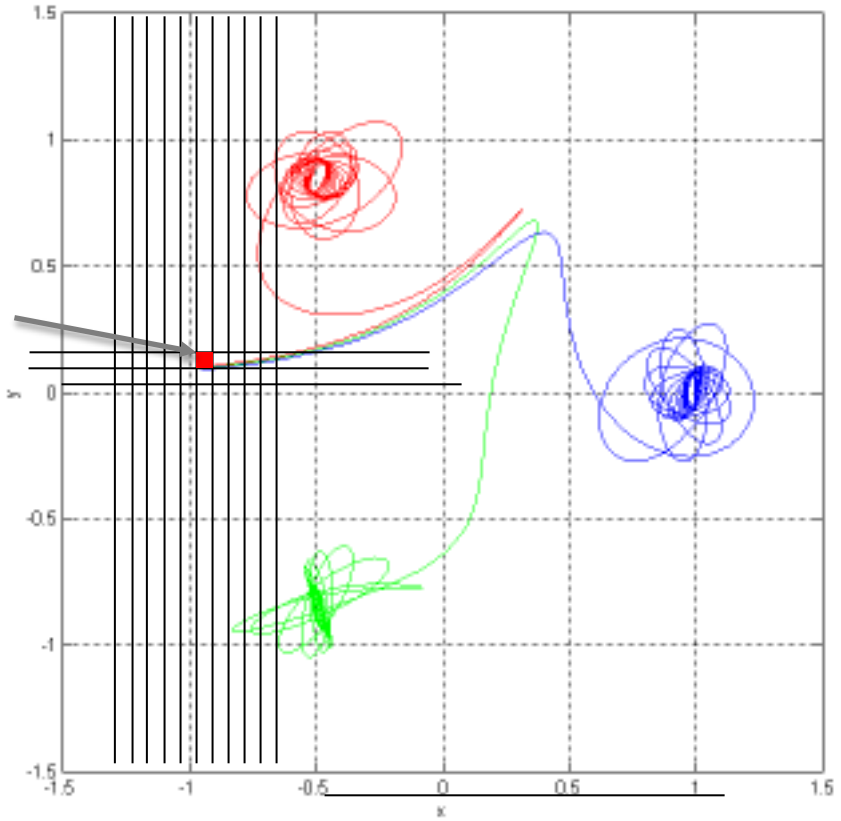


From Start Locations to an Image…

- **Idea:** Divide 2D plane into grid of pixels

1. For each pixel in the image, place the pendulum over pixel and let it go.
2. Simulate the trajectory of the pendulum until it comes to rest over one of the three magnets.
3. Color the pixel with the color of the magnet over which the pendulum ends up.

Pendulum ends up over red magnet when started here.



- Observation: The computations of the trajectory of one pixel does not depend on the computation for another pixel. This problem is said to be trivially parallelizable.
- Parallelization: Assign each thread a number of rows to compute.

Parallel Computation

- Divide image into bands of rows and assign each band of row to a different thread.

```
public class ParallelFractal extends JFrame {
  …
  private static final int NUM_THREADS = 8;
  private int[] imageBuffer = new int[WIDTH_PIXEL * HEIGHT_PIXEL];

  private void startComputation() {
    final int rowsPerThread = HEIGHT_PIXEL/NUM_THREADS;
    for (int i = 0; i < NUM_THREADS; i++) {
      final int tid = i;
      Thread thread = new Thread(() -> { // fork off compute threads
        for (int row = tid*rowsPerThread; row < (tid+1)*rowsPerThread && row < HEIGHT_PIXEL; row++) {
          double y = (HEIGHT_PIXEL/2 - row) * DY + Y_CENTER;
          for (int column = 0; column < WIDTH_PIXEL; column++) {
            double  x = (column - WIDTH_PIXEL/2) * DX + X_CENTER;
            int color = simulation.computeColorForPixel(x, y);
            imageBuffer[row * WIDTH_PIXEL + column] = color;
          }
        }
      });
      thread.start();
    }
  }
  ...
}
```

Array that will contain the pixel data of the image, i.e., a color for each pixel. Threads will write their results to this array.

Start *NUM_THREADS* thread, each Responsible to compute `rowsPerThread` rows.

Rows for which thread `tid` is responsible

Compute (x,y) coordinates in simulation coordinate system for pixel at `row` row and column `col`.

Update pixel in image buffer with color of magnet.

Visualization (1)

- User Interface Thread redraws image from **imageBuffer** data every 100 ms.
- User Interface Thread controlled via **Timer** (javax.swing.Timer), redraw when timer fires.

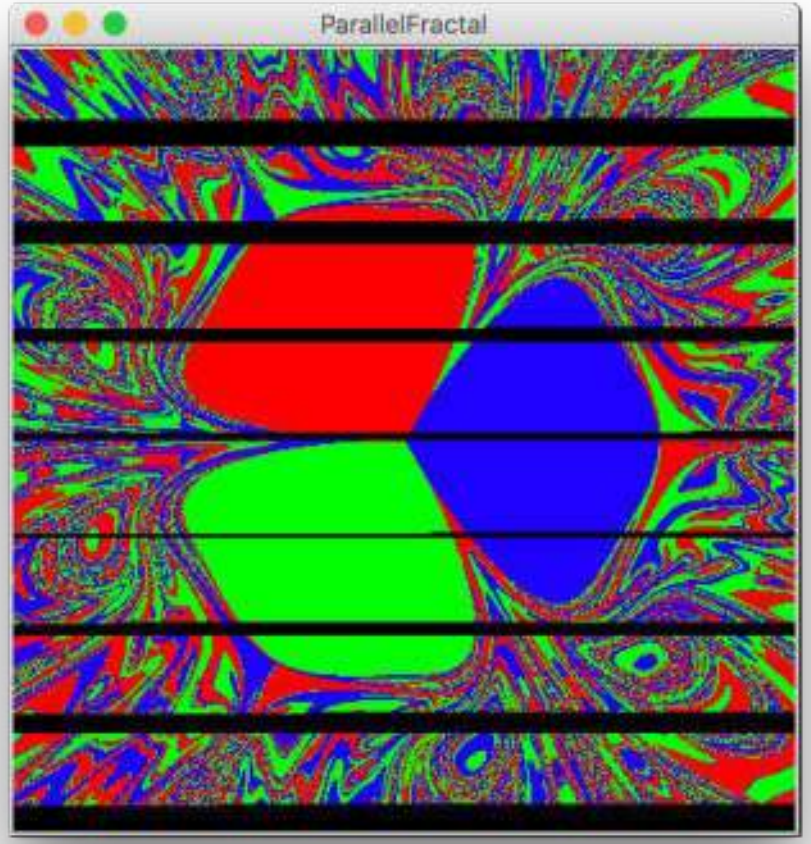
```
public class ParallelFractal extends JFrame {
  ...
  private Simulation simulation = new Pendulum();
  private Timer timer;
  public ParallelFractal(String title) {
    super(title);
    final BitmapPanel bitmap = new BitmapPanel();
    timer = new Timer(REDRAW_INTERVAL_MILLISECS, (event) ->  {
      bitmap.repaint();
    });
    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    getContentPane().add(new JScrollPane(bitmap));
    pack();  // Pack layout
  }
  ...
  public static void main(String[] args) {
    ParallelFractal app = new ParallelFractal("ParallelFractal");
    app.setVisible(true);
    app.startComputation();
  }
  ...
```

Visualization (2)

```
  ...
  /** The Bitmap Panel is the Swing component that draws the image.*/
  class BitmapPanel extends JPanel {

    BufferedImage image = new BufferedImage(WIDTH_PIXEL, HEIGHT_PIXEL, ColorSpace.TYPE_RGB);

    @Override
    public Dimension getPreferredSize() {
      return new Dimension(WIDTH_PIXEL, HEIGHT_PIXEL);
    }

    protected void paintComponent(Graphics g) {
      super.paintComponent(g);
      image.setRGB(0, 0, WIDTH_PIXEL, HEIGHT_PIXEL, imageBuffer, 0, WIDTH_PIXEL);
      g.drawImage(image, 0, 0, this);
    }
  }
}
```

Running the Program

400x400 Pixel Image



Thread 1

Thread 2

Thread 3

Thread 4

Thread 5

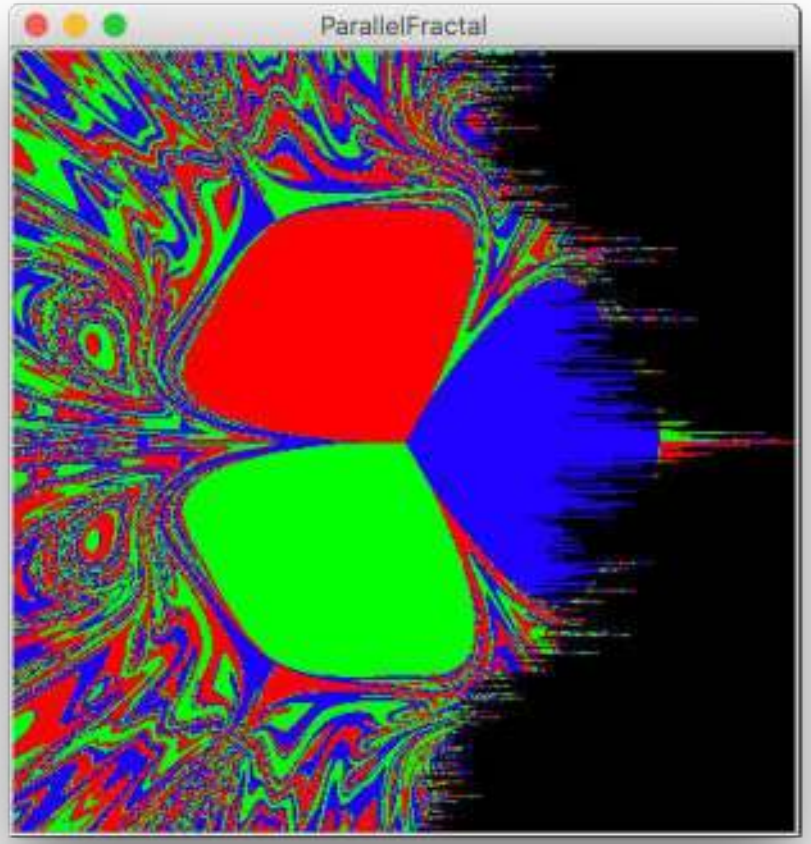
Thread 6

Thread 7

Thread 8

Threads make progress at different speeds.
Threads 4 + 5 finish faster than thread 1 and 8.
Reason: Different computation time in the physical simulation.
→ Degree of CPU utilization will decrease over time as threads finish their assigned pixels.

With 400 threads!



Java 1.8.0_162 on MacBook Pro 15" (2017)
CPU: i7-7920HQ (Kaby Lake), 4 physical cores,
8 logical cores.

Performance Analysis

Java 1.8.0_162 on MacBook Pro 15" (2017)
CPU: i7-7920HQ (Kaby Lake), 4 physical cores,
8 logical cores.

Threads	Seconds	Speedup (single-threaded)
1	251.2	1.0
4	69.9	3.6
8	39.5	6.4
16	36.3	6.9
40	36.1	7.0
400	35.4	7.1

- #threads < #logical cores results in uneven distribution of work. Some threads will finish earlier and some cores will remain idle.
- #threads >> #logical cores solves alleviates "imbalance" problem, however, overheads increase due to thread switching.
- Better solution: work-stealing
  Idea: Threads that are finished with their own work steal work from other busy threads.
- **Interesting observation**: Despite being a 100% compute dominated program, *performance improves if the number of threads is increased beyond the number of logical cores*.

Performance Comparison

- Same 400 x 400 Pixel image

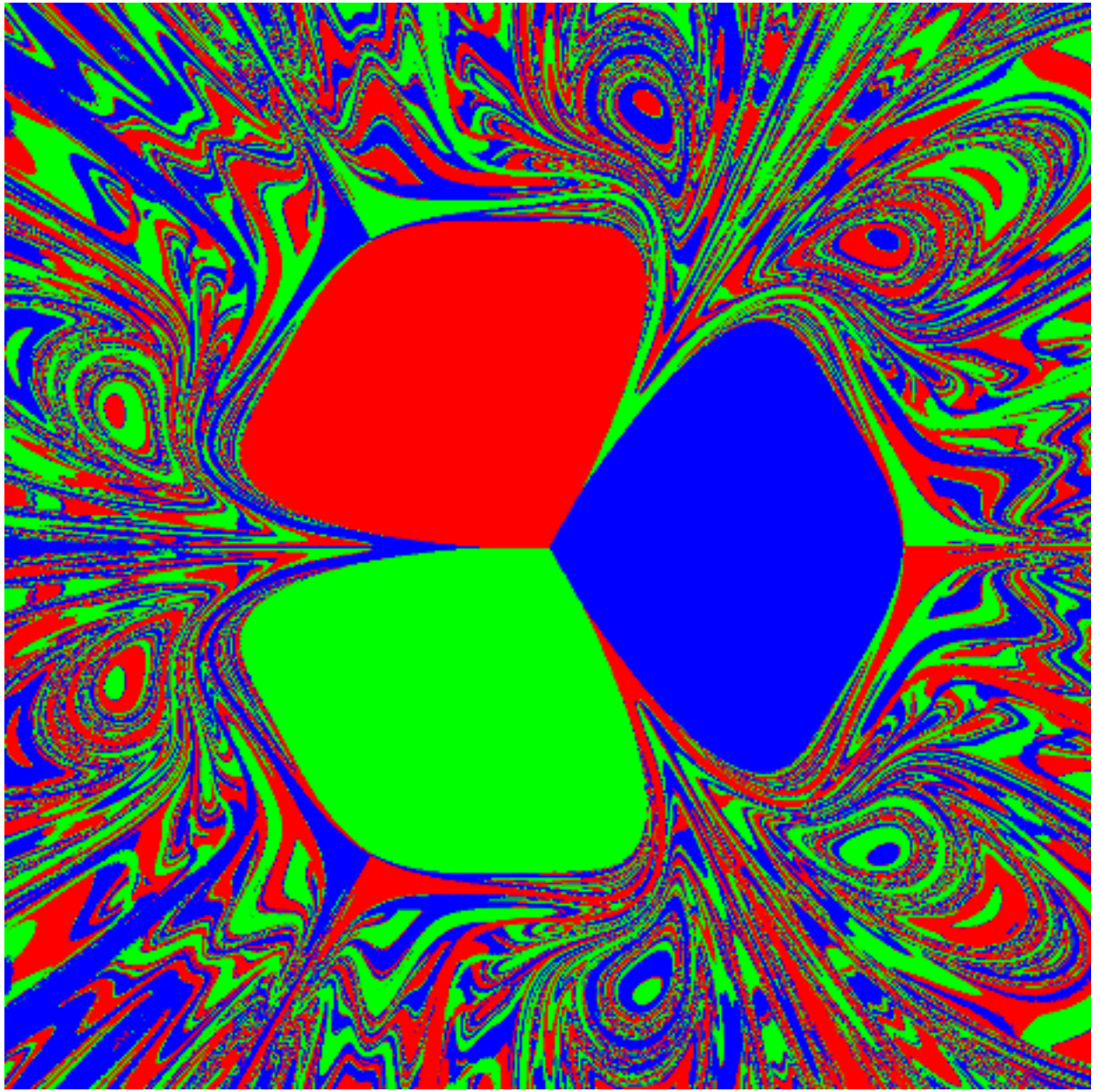
Silicon Graphics Workstation released in 1996

System	CPU	Execution Time (wall clock)	Remarks
SGI O2 Workstation	MIPS R5000 200 MHz (single CPU)	189 min	Linux, gcc 4.3.2, single threaded
PowerMac G5 Quad	2xdual-core PPC970MP, 2.5GHz	2 min 45 sec	Linux, gcc
IBM Power 795	32x8-core POWER7, 4GHz	1.55 sec	AIX 7.1, IBM C/C++ Compiler, 256 threads
Kaby Lake	1x4-core Kaby Lake i7-7700K, 4.2GHz	19.45 sec	Linux, gcc 5.4.0 -mavx2, 8 threads
Tesla K40c in IBM Power S824	15 x 192 cores = 2,880 cores	1.22 sec	CUDA7.0, Linux, (60x192)
MSI GTX 1080	20 x 128 cores = 2,560 cores	4.415 sec	CUDA8.0, Linux, (80x128)
MSI GTX 1080	20 x 128 cores = 2,560 cores	0.063 sec	CUDA8.0, Linux, **single-precision arithmetic**, (40x512)

GPUs

Best Java on MacBook Pro 15" (4 core Kaby Lake) is about 36 seconds.

Final Image (1000x1000)



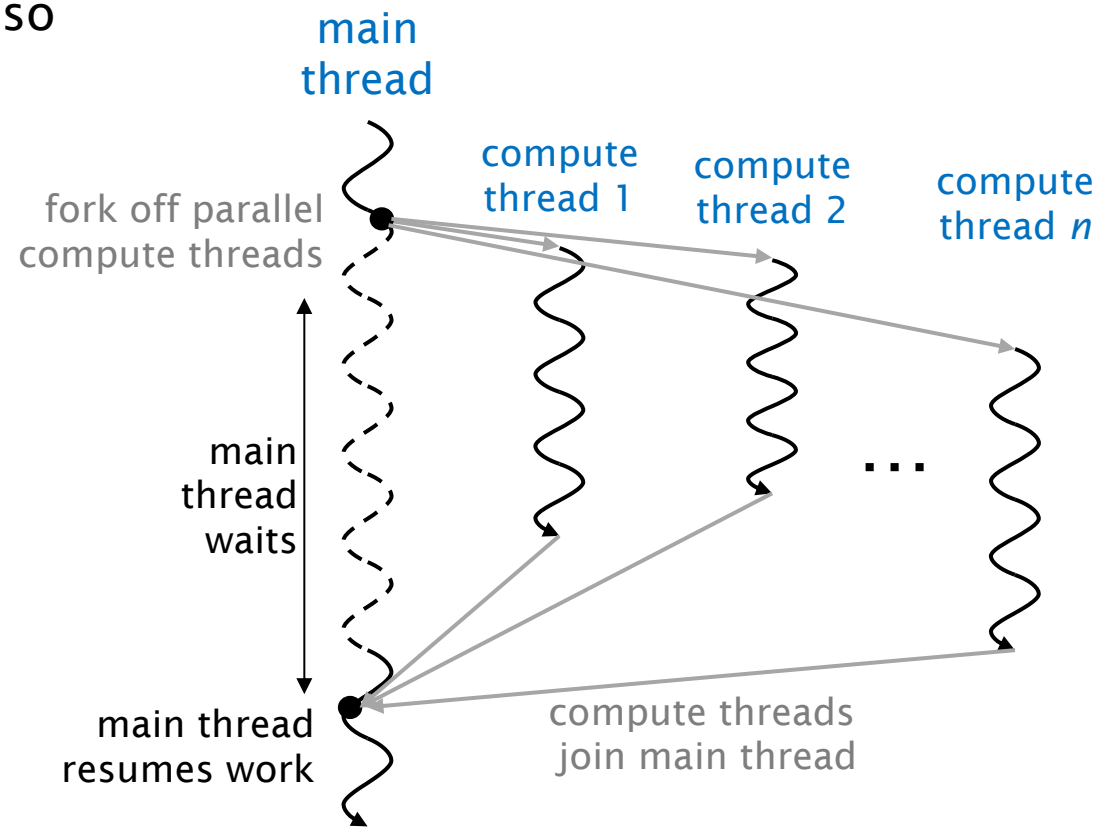
Single-Precision Arithmetic

computed on MacBook Pro 15"
using the AMD Radeon Pro 560
via OpenCL in 1.3 seconds.

Forking off Threads and Joining them Afterwards

- Parallelizing computation is difficult. Very often, not all steps of a computation can be parallelized (or doing so may be inefficient).
  → Computation consists of parallel and sequential components.

- Fork-Join Compute Model
  1. Sequential (single-threaded) computation at the beginning
  2. Parallel computation, forking off threads from main thread
  3. Parallel threads join main thread
  4. Main thread resumes single-threaded computation



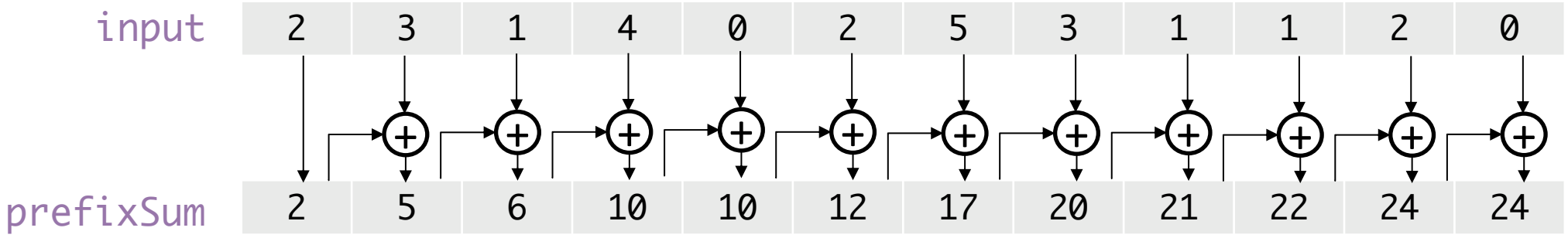
Parallelizing Prefix Sum Computation (from earlier)

- Compute the prefix sum in parallel

Sequential Code:

```
static int[] input = ...;
static long[] prefixSum = ...;

static void sequentialPrefixSum() {
  prefixSum[0] = input[0];
  for (int j = 1; j < LENGTH; j++) {
    prefixSum[j] = prefixSum[j - 1] + input[j];
  }
}
```

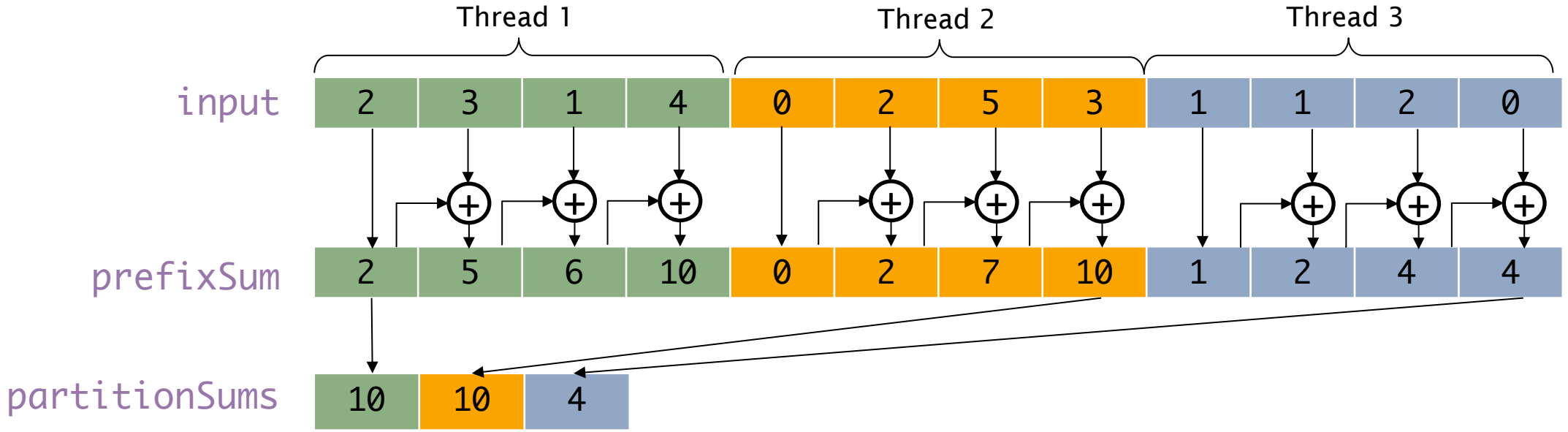
Loop dependency!
Iteration j requires on data computed by previous iteration j-1.



Prefix Sum: Phase 1 – Prefix sum on Partitions (parallel)

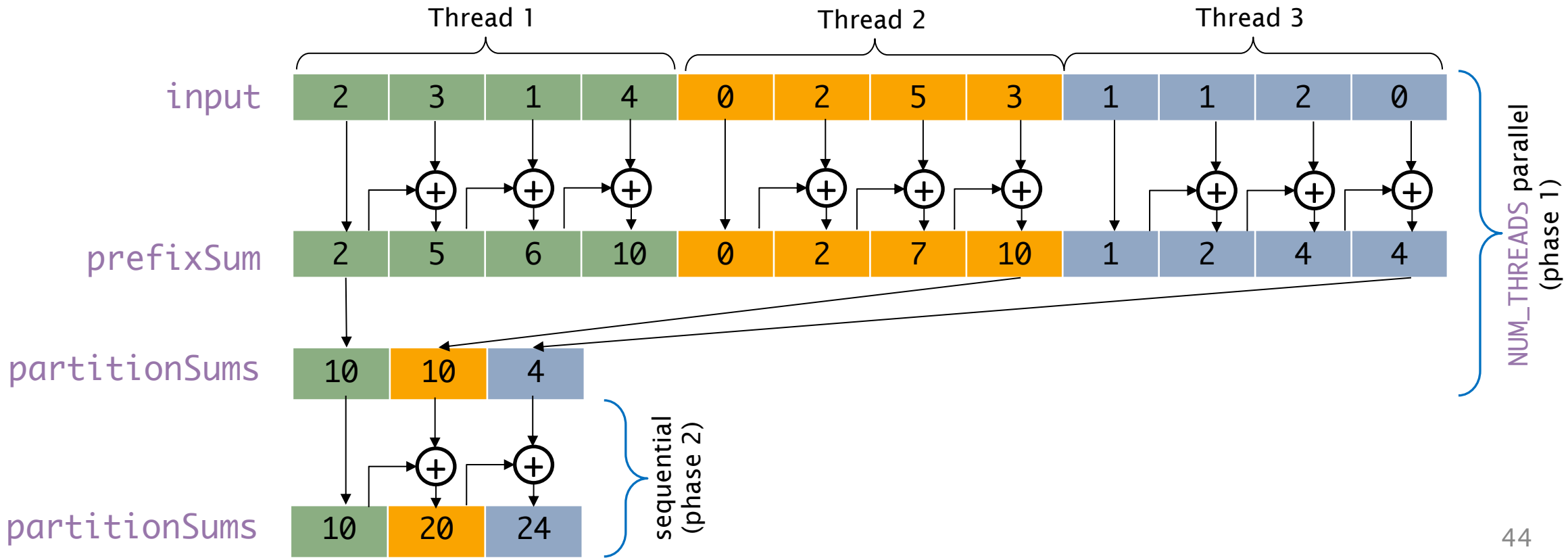
- Partition array `input` among threads.
- Each thread computes the prefix sum on its assigned partition (assume 1 partition/thread).
- Once with its partition, each thread writes its last sum value into the array `partitionSums`.
  - Array `partitionSums` contains the last sum value for each partition.

```
static int[] input = ...;
static long[] prefixSum = ...;
static long[] partitionSums = new long[NUM_THREADS];
```



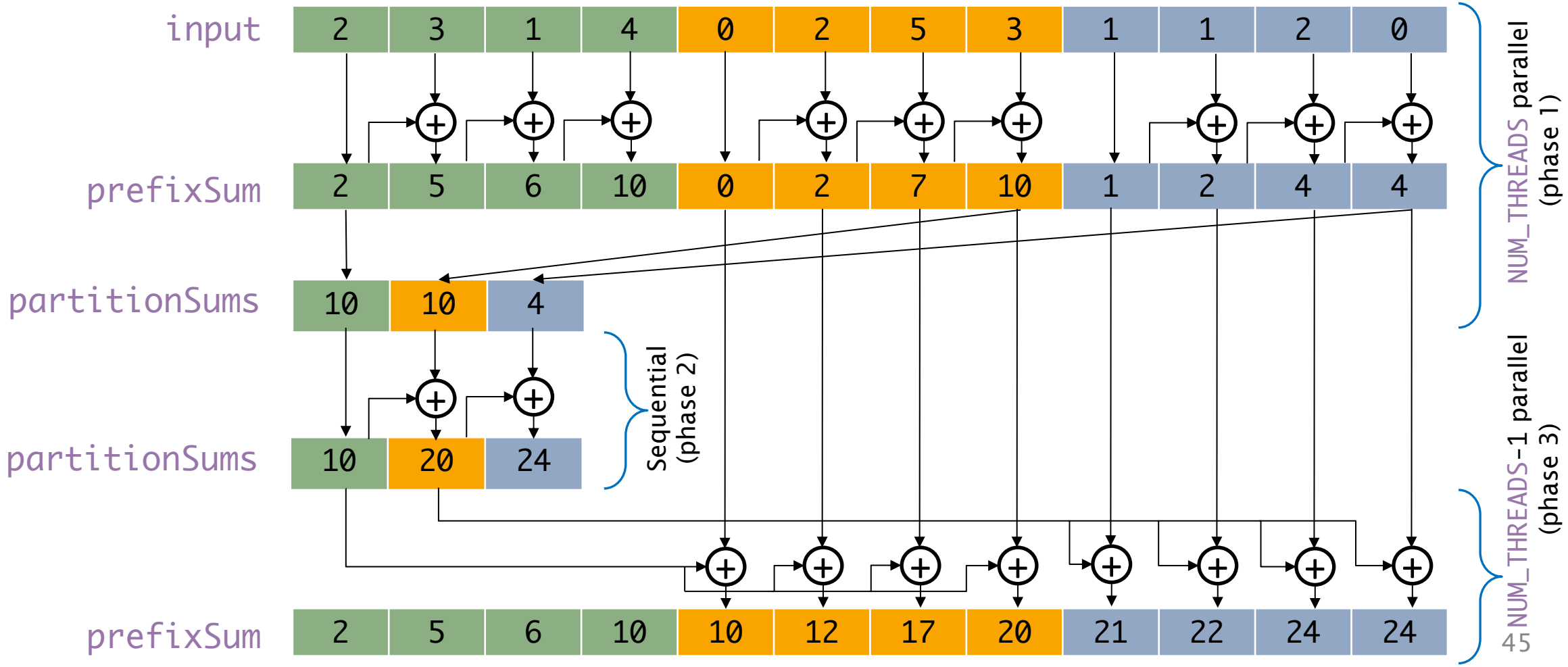
Prefix Sum: Phase 2 – Prefix Sum on Partition Sums (sequential)

- Parallel compute threads are done.
- Main thread resumes and computes prefix sum `partitionSums` array.
  - Instead of `LENGTH` elements, only `NUM_THREADS` elements need to be summed up by the main thread. Performance gain if `LENGTH` >> `NUM_THREADS`.

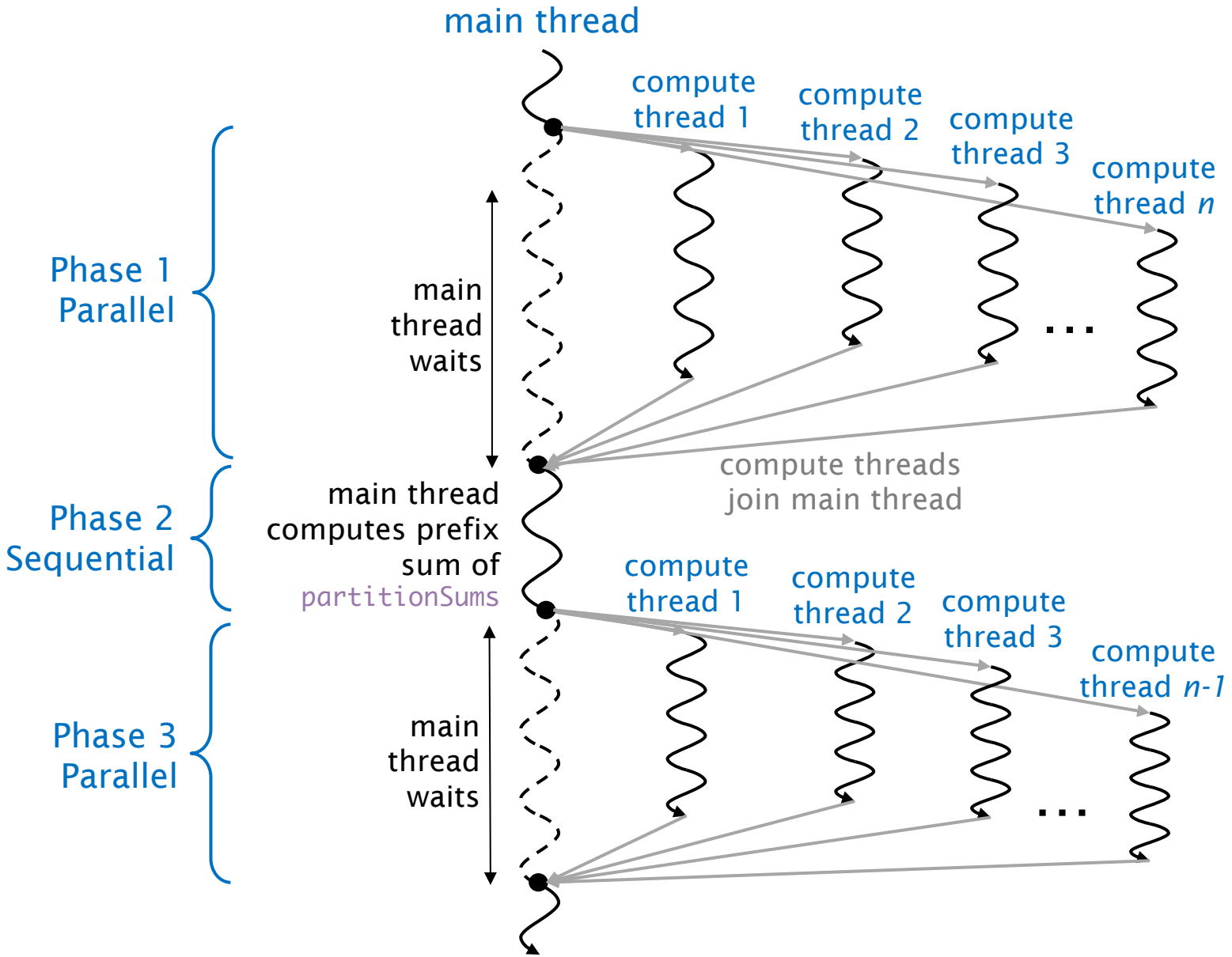


Prefix Sum: Phase 3 – Add Partition Offset to Partitions (parallel)

- Fork off NUM_THREADS-1 parallel compute threads that will add partitionSums as offsets to all elements of their partition.
  - The first partition does need any updates (its offset is zero).



Prefix Sum: 2 Parallel Phases – 1 Sequential Phase



Download ParallelPrefixSum.java from Moodle.

Prefix Sum in Java (1)

Java Program:

1. Fill `input` with random integers
2. Compute `prefixSum`

```
public class ParallelPrefixSum {
  private static final int LENGTH = 100000000;
  private static final int NUM_THREADS = 4;

  private static int[] input = new int[LENGTH];
  private static long[] prefixSum = new long[LENGTH];

  /** Sequential implementation for reference. */
  private static void sequentialPrefixSumGenRandom() {
    Random random = new Random();
    long sum = 0;
    for (int j = 0; j < LENGTH; j++) {
      int newValue = random.nextInt();      // generate new random number
      sum += newValue;                      // update cumulative sum
      input[j] = newValue;
      prefixSum[j] = sum;
    }
  ...
}
```

Prefix Sum in Java (2)

```
public class ParallelPrefixSum {
  private static long[] partitionSums = new long[NUM_THREADS];
  ...
  private static void parallelPrefixSumGenRandom() {
    Thread[] threads = new Thread[NUM_THREADS];
    final int elementsPerThread = LENGTH / NUM_THREADS;

    // Phase 1 (PARALLEL): prefix sum over partitions
    for (int i = 0; i < NUM_THREADS; i++) {
      final int tid = i;
      threads[i] = new Thread(() -> {
        int myStartIdx = tid * elementsPerThread;      // determine start index for thread tid
        Random random = new Random();
        long mySum = 0;
        for (int j = 0; j < elementsPerThread; j++) {  // iterate over thread's partition of the data
          int newValue = random.nextInt();             // generate new random number
          mySum += newValue;                           // update cumulative sum for this thread
          input[myStartIdx + j] = newValue;
          prefixSum[myStartIdx + j] = mySum;
        }
        partitionSums[tid] = mySum;
      });
      threads[i].start();
    }
  // main-thread joins all threads
  for (Thread t : threads) {
    try { t.join(); } catch (InterruptedException e) { e.printStackTrace(); }
  }
  ...
```

Main thread forks of compute threads

Main thread joins all compute threads. Blocks until compute threads are done.

Thread could be interrupted while waiting, so we need to catch an exception which might be thrown.

Prefix Sum in Java (3)

```
    ...
    // Phase 2 (SEQUENTIAL): main thread computes prefix sum on partitionSums
    for (int i = 1; i < NUM_THREADS; i++) {
      partitionSums[i] = partitionSums[i] + partitionSums[i - 1];
    }

    // Phase 3 (PARALLEL): update prefixSum with partition offsets
    // First partition does not need any work!
    for (int i = 1; i < NUM_THREADS; i++) {
      final int tid = i;
      threads[i] = new Thread(() -> {
        int myStartIdx = tid * elementsPerThread;      // determine start index for thread tid
        for (int j = 0; j < elementsPerThread; j++) {  // iterate over thread's partition of the data
          prefixSum[myStartIdx + j] += partitionSums[tid - 1]; // update with partition offset
        }
      });
      threads[i].start();
    }
    // main-thread joins all threads
    for (int i = 1; i < NUM_THREADS; i++) {
      try { threads[i].join(); } catch (InterruptedException e) { e.printStackTrace(); }
    }
  }


  ...
}
```

Prefix Sum in Java (4)

```
...
public static void main(String[] args) {
  System.out.println("Array length: " + LENGTH + " elements");
  String[] names = {"Sequential", "Parallel (" + NUM_THREADS + "-way)"};

  // Use functional interface Runnable to collect all implementations as an array.
  Runnable[] implementations = { ParallelPrefixSum::sequentialPrefixSumGenRandom,
    ParallelPrefixSum::parallelPrefixSumGenRandom }; // Method references

  for (int i = 0; i < implementations.length; i++) {
    long startTime = System.nanoTime();
    implementations[i].run();
    long endTime = System.nanoTime();
    assertPrefixSumCorrect(); // Check whether prefixSum is correct

  double seconds = (endTime - startTime) * 1e-9;
  System.out.format("%20s\t%f seconds\n", names[i], seconds);
}
...
```

Prefix Sum in Java (5)

```
  ...
  /** Checks whether prefixSum is the correct prefix sum of input. */
  private static void assertPrefixSumCorrect() {
    long sum = 0;
    for (int i = 0; i < input.length; i++) {
      sum += input[i];
      if (prefixSum[i] != sum) {
        throw new AssertionError("invalid prefix sum at " + i +
            ", expected " + sum + " got " + prefixSum[i] + ", aborting...\n");
      }
    }
  }

}
```

Running the Prefix Sum Program

- **Output**: (MacBook Pro 15", i7-7920HQ (Kaby Lake), 4 physical cores, 8 logical cores)

```
Array length: 100000000 elements
          Sequential   1.088395 seconds
    Parallel (4-way)   0.327048 seconds
```

- **Observation**
  - The speedup is 3.3x using 4-threads on the 4-core system.
  - The parallel code is much more complex and almost 6x as long (just compare it to sequential code).
    → 56 lines vs. 10 lines of code.

Thread Pools

- Creating and destroying threads is expensive.
- The overhead of creating and destroying threads may eat up the speedup gained by parallelization (especially when using many threads for small tasks)!
- Instead of always creating a new thread, reuse existing threads.
- Pool threads:
  - When a thread is finished put it to sleep and place it a pool.
  - When you need a thread, find one in the pool, assign it new code to run and wake it up.

Thread Pools

- A Thread Pool contains n of threads to perform m tasks concurrently.
  - Typically, $n \neq m$.
  - For fixed size thread pools, n is chosen as depending on the computing resources available (cores, memory) or the expected compute:wait-time ratio of the threads when performing I/O.
  - m depends on the problem size.
  - The Thread Pool Executor assigns the m tasks to the n threads. If $m > n$, some tasks will have queued.

Thread Pools in Java

- Thread Pools are represented by as an `ExecutorService` (interface).
- The are different implementations of Thread Pools available in the package `java.util.concurrent`.

`FixedThreadPool`	Thread pool with a fixed number of threads *n* and one shared unbounded task queue. There are never more than *n* threads in the pool. The thread are allocated lazily.
`SingleThreadExecutor`	Thread pool with one single thread and one shared unbounded task queue. Similar to a `FixedThreadPool` with *n* = 1, but the thread pool size cannot be changed afterwards (via hack).
`CachedThreadPool`	Creates new a thread if a task is submitted while all threads in the pool are busy. Idle threads remain in the pool for 60 seconds until they are removed.
`WorkStealingPool`	A thread pool that maintains enough threads to support the given parallelism level (explicitly specified or #logical cores), and may use multiple queues to reduce contention. No guarantee of execution order of the task.
`ScheduledThreadPool`	A thread pool of *n* threads that can schedule commands to run after a given delay, or to execute periodically.
`SingleThreadScheduled Executor`	A single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.

- The Thread Pools can be created using the static methods `new...` in the Factory class `Executors`.

The ExecutorService Interface

Methods of the ExecutorService Interface:

- `void execute(Runnable command)`: Execute the specified runnable (like in a Thread) without any notification about its completion.
- `<T> Future<T> submit(Runnable task, T result)`: Execute the specified Runnable (like in a Thread) and return the specified result in a Future that completes when the task is done.
- `Future<?> submit(Runnable task)`: Execute the specified Runnable (like in a Thread) and return a Future that completes with the value null when the task is done.
- `<T> Future<T> submit(Callable<T> task)`: Execute the specified Callable that produces a result of type T and return it as a Future.
- `void shutdown()`: initiated orderly shutdown of thread pool. Previously submitted tasks will be executed but not new tasks will be accepted.
- `boolean isShutdown()`: returns true if the executor has been shut down.
- `boolean isTerminated()`: returns true if all tasks have completed following shut down.
- `boolean awaitTermination(long timeout, TimeUnit unit)`: Blocks until all tasks have completed execution after a shutdown.

Example: Fixed-size Thread Pool

```
public class ThreadPoolExample {

  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(4);  // pool with 4 threads

    for (int i = 0; i < 13; i++) {  // submit a number of tasks
      final int taskid = i;
      executor.execute(() -> {
        System.out.println(LocalDateTime.now() + ": start task " + taskid);
        int thinkingTimeMillis = new Random().nextInt(10000);
        try { Thread.sleep(thinkingTimeMillis); } catch (InterruptedException ignored) { }
        System.out.println(LocalDateTime.now() + ": finished task " + taskid);
      });
    }

    executor.shutdown();
    try {              // wait until all tasks finished and executor has terminated
       executor.awaitTermination(1, TimeUnit.MINUTES);  // but no longer than 1 min
    } catch(InterruptedException ignored) { }
    System.out.println("All tasks have been completed.");
  }
}
```

Example: Fixed-size Thread Pool: Output

```
2018-05-15T21:57:55.477: start task 3
2018-05-15T21:57:55.477: start task 0
2018-05-15T21:57:55.477: start task 2
2018-05-15T21:57:55.477: start task 1
2018-05-15T21:57:59.940: finished task 3
2018-05-15T21:57:59.940: start task 4
2018-05-15T21:58:00.246: finished task 0
2018-05-15T21:58:00.246: start task 5
2018-05-15T21:58:01.226: finished task 5
2018-05-15T21:58:01.226: start task 6
2018-05-15T21:58:02.096: finished task 1
2018-05-15T21:58:02.096: start task 7
2018-05-15T21:58:03.174: finished task 2
2018-05-15T21:58:03.175: start task 8
2018-05-15T21:58:04.394: finished task 6
2018-05-15T21:58:04.394: start task 9
2018-05-15T21:58:05.494: finished task 4
2018-05-15T21:58:05.495: start task 10
2018-05-15T21:58:06.451: finished task 8
2018-05-15T21:58:06.451: start task 11
2018-05-15T21:58:06.627: finished task 9
2018-05-15T21:58:06.628: start task 12
2018-05-15T21:58:07.385: finished task 7
2018-05-15T21:58:12.495: finished task 11
2018-05-15T21:58:13.306: finished task 10
2018-05-15T21:58:15.457: finished task 12
All tasks have been completed.
```

After all 4 worker threads were assigned a task, new tasks only start after a previously running task finished and a thread becomes available.

Download ParallelPrefixSumThreadPool.java from Moodle.

Parallel Prefix Sum with Work-Stealing Thread Pool (size #log. cores)

```
public class ParallelPrefixSumThreadPool {

  private static final int LENGTH = 100000000;
  private static final int NUM_PARTITIONS = 1000;

  private static int[] input = new int[LENGTH];
  private static long[] prefixSum = new long[LENGTH];
  private static long[] partitionSums = new long[NUM_PARTITIONS];

  private static ExecutorService executor = Executors.newWorkStealingPool();

  public static void main(String[] args) {
    ... // trivial changes, essentially unmodified
    executor.shutdown();
  }

  private static void parallelPrefixSumGenRandom() { ... }

  private static void sequentialPrefixSumGenRandom() { … }

  private static void assertPrefixSumCorrect() { ... }
}
```

Specify number of partitions instead number of threads.

Parallel Prefix Sum with Work-Stealing Thread Pool (2)

```
private static void parallelPrefixSumGenRandom() {
  final int elementsPerPartition = LENGTH / NUM_PARTITIONS;
  Future<?>[] completions = new Future[NUM_PARTITIONS];

  // Phase 1 (PARALLEL): prefix sum over partitions
  for (int i = 0; i < NUM_PARTITIONS; i++) {
    final int pid = i;    // partition ID
    completions[i] = executor.submit(() -> {
      int myStartIdx = pid * elementsPerPartition;      // determine start index for partition pid
      Random random = new Random();
      long mySum = 0;
      for (int j = 0; j < elementsPerPartition; j++) {  // iterate over the partition
        int newValue = random.nextInt();               // generate new random number
        mySum += newValue;                             // update cumulative sum for this partition
        input[myStartIdx + j] = newValue;
        prefixSum[myStartIdx + j] = mySum;
      }
      partitionSums[pid] = mySum;
    });
  }
  // main-thread collects all completion Futures to make sure
  // that all partitions are processed
  for (Future<?> f : completions) {
    try { f.get(); } catch (Exception e) { e.printStackTrace(); }
  }
  ...
```

Use an array of Futures that hold the completion "signals" of the tasks. This allows the main thread to figure out when all the parallel phases are complete.

Main thread retrieves the values the Futures of the tasks.

The thread will block on `f.get()` until the value of the Future `f` becomes available, i.e., the corresponding task has finished. The value of the future (`null`) is ignored. `f.get()` is just used for synchronization.

Parallel Prefix Sum with Work-Stealing Thread Pool (3)

```
  ...
  // Phase 2 (SEQUENTIAL): main thread computes prefix sum on partitionSums
  for (int i = 1; i < NUM_PARTITIONS; i++) {
    partitionSums[i] = partitionSums[i] + partitionSums[i - 1];
  }

  // Phase 3 (PARALLEL): update prefixSum with partition offsets
  // First partition does not need any work!
  for (int i = 1; i < NUM_PARTITIONS; i++) {
    final int pid = i;  // partition ID
    completions[i] = executor.submit(() -> {
      int myStartIdx = pid * elementsPerPartition;      // determine start index for partition pid
      for (int j = 0; j < elementsPerPartition; j++) {  // iterate over the partition
        prefixSum[myStartIdx + j] += partitionSums[pid - 1]; // update with partition offset
      }
    });
  }
  // main-thread collects all completion Futures to make sure that all partitions are processed
  for (int i = 1; i < NUM_PARTITIONS; i++) {
    try {
      completions[i].get();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Output

```
Array length: 100000000 elements
          Sequential  1.022934 seconds
            Parallel  0.291947 seconds
```

The ScheduledExecutorService Interface

The `ScheduledExecutorService` contains some additional methods that allow specifying the time instance a task should be scheduled:

- `ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)`: Executes the specified command once after the specified delay. Returns a `Future` that informs about the completion of the command (will return `null`).
- `ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)`: Executes specified callable once after the specified delay. Returns a `Future` that will yield the value that is returned by the `call()` method of `Callable`).
- `ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`: Executes specified callable many times with `delay unit` delay between invocations after the `initialDelay`. Returns a `Future` that informs about the completion of the command (will return `null`).
- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`: Executes specified callable many times with `delay unit` delay after the termination of the previous invocation and an `initialDelay` initially. Returns a `Future` that informs about the completion of the command (will return `null`).

Example: Scheduled Thread Pool Executor

```java
public class ScheduledThreadPoolExample {

  public static void main(String[] args) {
    ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

    final long startTime = System.currentTimeMillis();
    executor.scheduleAtFixedRate(
      () -> System.out.println((System.currentTimeMillis()-startTime) + " task @fixed rate"),
      3 /* initialDelay */, 1 /* period */, TimeUnit.SECONDS);

    executor.schedule(
      () -> System.out.println((System.currentTimeMillis()-startTime) + " task @5s"),
      5 /* delay */, TimeUnit.SECONDS);

    executor.schedule(() -> executor.shutdown(), 10 /* delay */, TimeUnit.SECONDS);

    try { // Main thread waits until all tasks have
      // finished and executor terminated.
      executor.awaitTermination(1, TimeUnit.MINUTES);
    } catch(InterruptedException ignored) { }
  }

}
```

Output

```
3056 task @fixed rate
4058 task @fixed rate
5056 task @fixed rate
5056 task @5s
6056 task @fixed rate
7054 task @fixed rate
8056 task @fixed rate
9054 task @fixed rate
10054 task @fixed rate
```

Exercise 3: Problem 1

Parallel computation of cryptographic file hash SHA-256 (digest) using thread pool.

Exercise 3: Problem 2

Pendulum Fractal with a Work-Stealing Thread Pool

Race Conditions

Data Conflicts between Threads

Download DataRace.java from Moodle

Threads Updating a Shared Variable

```
public class DataRace {

  private static final int NUM_THREADS = 2;
  private static final int PER_THREAD_COUNT = 100000;
  private static int counter = 0;

  public static void main(String[] args) {
    Thread[] threads = new Thread[NUM_THREADS];
    for (int i = 0; i < threads.length; ++i) {
      threads[i] = new Thread(() -> {
        for (int j = 0; j < PER_THREAD_COUNT; j++) {
          counter++;
        }
      });
      threads[i].start();
    }
    for (Thread t : threads) {
      try { t.join(); } catch (InterruptedException e) { e.printStackTrace(); }
    }
    System.out.println("counter = " + counter + "  expected " + (PER_THREAD_COUNT * NUM_THREADS) +
        " ("+NUM_THREADS+" thread(s)");
  }

}
```

Output

```
counter = 106123  expected 200000 (2 threads)
```

```
counter = 182563  expected 800000 (8 threads)
```

Incorrect results for all but 1 thread!

```
counter = 100000  expected 100000 (1 threads)
```

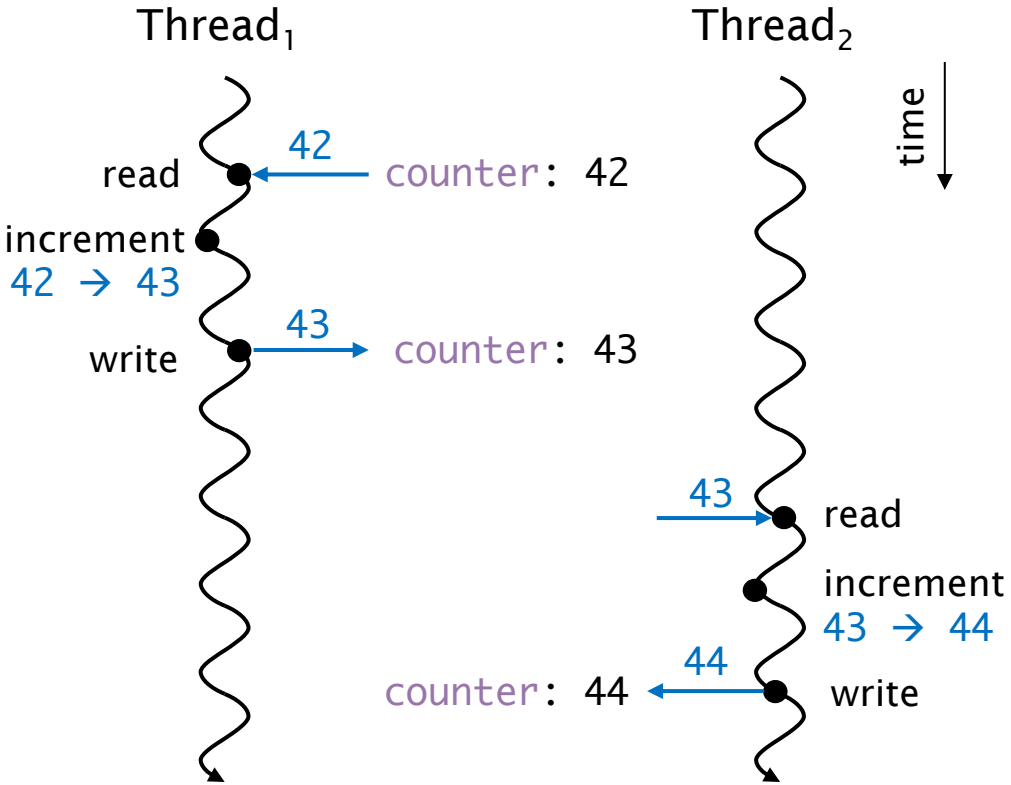
Multiple threads read and update the shared variable counter.

We seem to loose updates!

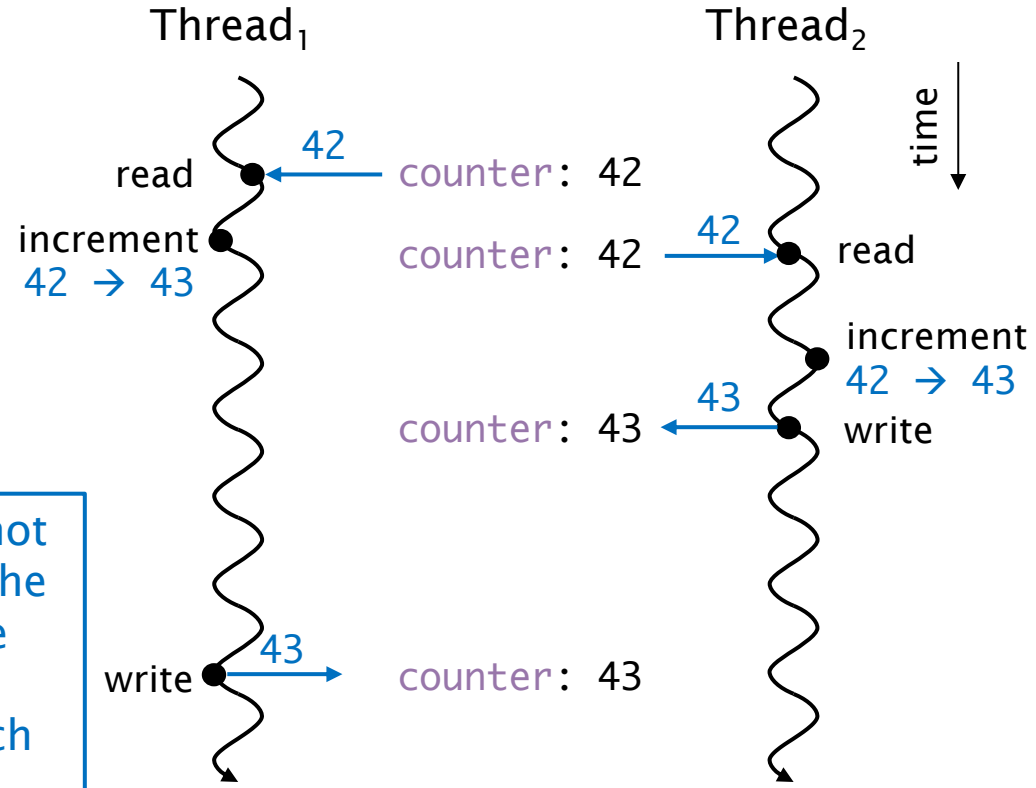
Serialized Execution Schedules

Consider the following two execution schedules for NUM_THREADS = 2:

"Good" Schedule



"Bad" Schedule



In general, we do not have control over the execution schedule as the threads run independent of each other.
"Bad" Schedules can always happen!

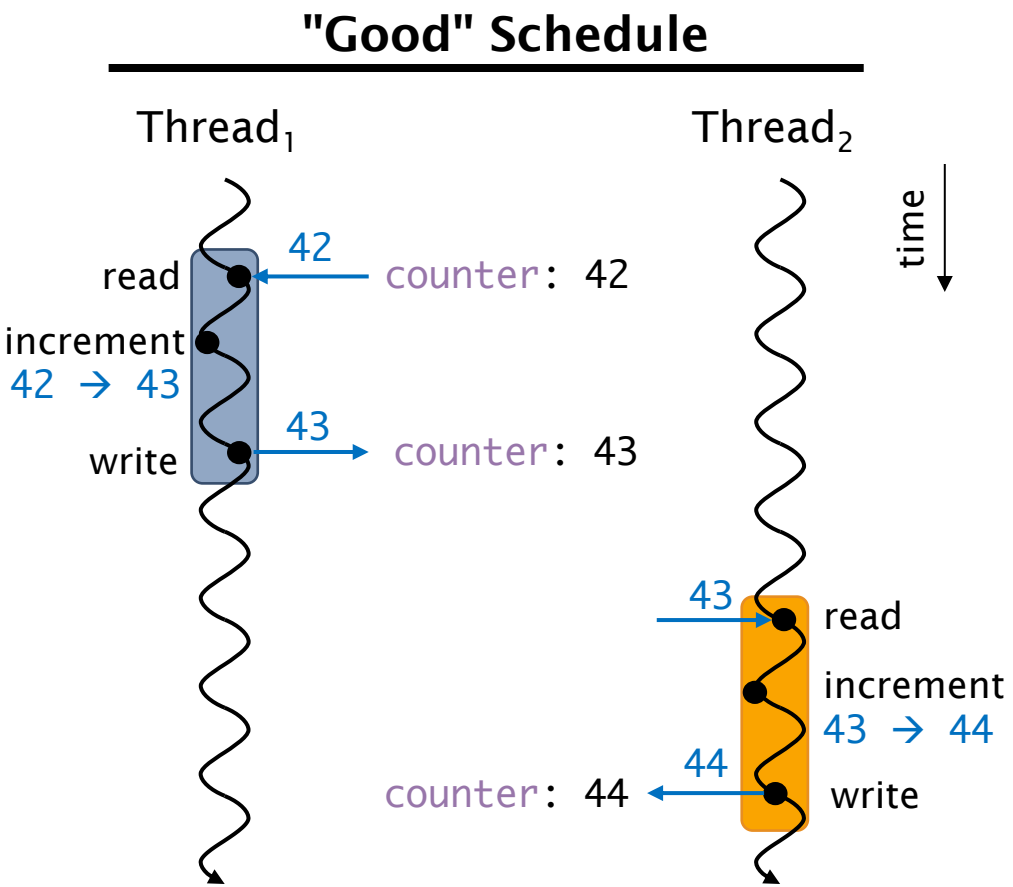
Update of Thread$_2$ was lost, it got overwritten by Thread$_1$.

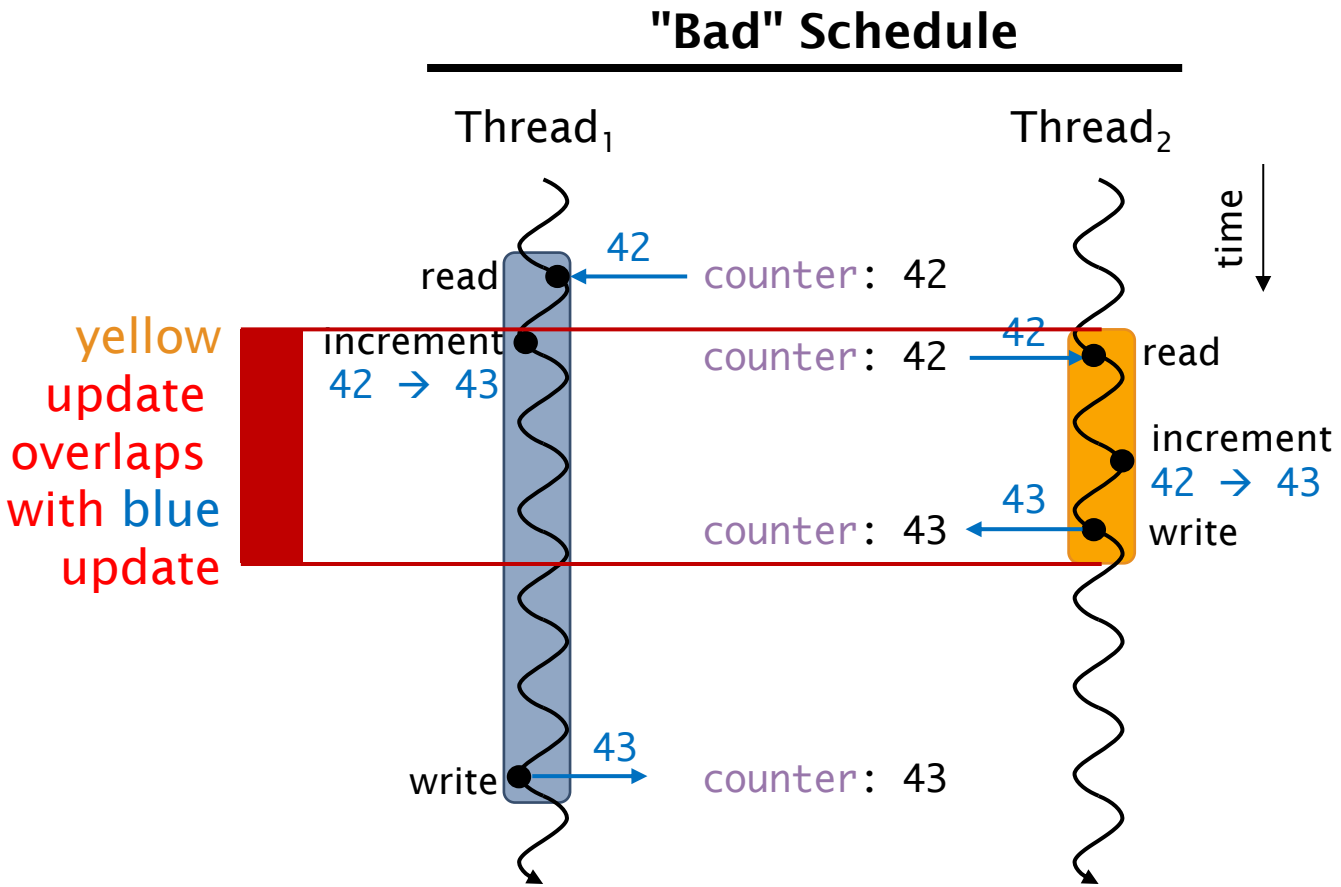
Race Conditions

- When threads update shared variables the order of the writes is not deterministic.
- The relative ordering of writes can affect the program's correctness!
  - This is a bad situation and is called a Race Condition.
  - Term originates from digital electronics. It refers to the a situation in which the outcome depends on the relative timing of events which may be uncontrollable (Race: whoever comes first, wins).

- Race Conditions are extremely difficult to debug.
  - Bugs appear non-deterministically.
  - Bugs due to "bad" schedules could happen only very rarely and hard to reproduce.

Analysis: Where was the Problem?





→ **Solution: we need to keep the read-and-write together such that no other update can come in-between.**

Atomic Updates

- Requirement: The Read/Increment/Write-Update must be atomic.
  - From Greek átomos ἄτομος = *indivisible* (ἄ = *not*, τέμνω = *I cut*)

- **Atomic Operations**
  - Set of operations that either have all happened or none.
  - No thread can see an intermediate state in which only part of the operations happened.

- For the read/increment/write this means that no thread can jump between the read and the write.
- This prevents the overlapping "bad" schedule.

Atomic Instructions

- Java provides classes that support atomic operations (**java.util.concurrent.atomic**).
  - These operation will translate to the corresponding atomic CPU instructions (e.g., `lock xadd`).

- Class **AtomicInteger**:

```
AtomicInteger counter = new AtomicInteger(0);  // create new atomic integer with initial value 0

int val1 = counter.getAndIncrement();   // returns old value (0) and increments counter (0 -> 1)
int count1 = counter.get();             // returns current value (1)

int val2 = counter.incrementAndGet();   // increments counter (1 -> 2) and returns new value (2)
int count2 = counter.get();             // returns current value (2)

int val3 = counter.getAndAdd(4);        // returns old value (2) and adds 4 to counter (2 -> 6)
int count3 = counter.get();             // returns current value (6)

int val4 = counter.addAndGet(2);        // returns new value (8) and adds 2 to counter (6 -> 8)
int count4 = counter.get();             // returns current value (8)
```

- Similar classes **AtomicLong**, **AtomicBoolean**, …

Threads Updating a Shared Variable Atomically

```
public class DataRace {

  private static final int NUM_THREADS = 2;
  private static final int PER_THREAD_COUNT = 100000;
  private static AtomicInteger counter = new AtomicInteger(0);

  public static void main(String[] args) {

    Thread[] threads = new Thread[NUM_THREADS];
    for (int i = 0; i < threads.length; ++i) {
      threads[i] = new Thread(() -> {
        for (int j = 0; j < PER_THREAD_COUNT; j++) {
          counter.incrementAndGet();
        }
      });
      threads[i].start();
    }
    for (Thread t : threads) {
      try { t.join(); } catch (InterruptedException e) { e.printStackTrace(); }
    }
    System.out.println("counter = "+counter+"  expected " + (PER_THREAD_COUNT * NUM_THREADS) +
        " ("+NUM_THREADS+" threads)");
  }
}
```

Output

```
counter = 200000  expected 200000 (2 threads)
```

```
counter = 800000  expected 800000 (8 threads)
```

→ Correct Result.
→ No updates lost.

Mutual Exclusion

Preventing Race Conditions

Download `GroupCount.java` from Moodle

Race Conditions in more complex scenarios

Example: Given key/value pairs, compute average of values for every key

```
public class GroupCount {
  private static class Aggregate {
    AtomicInteger count = new AtomicInteger(0); // use atomic integer for safe updates
    AtomicInteger sum = new AtomicInteger(0);   // use atomic integer for safe updates
  }
  private final Map<Integer, Aggregate> counts = new HashMap<>();

  public void add(int key, int value) {
    Aggregate agg = counts.get(key);
    if (agg == null) {
      agg = new Aggregate(); // insert if not already in map
      counts.put(key, agg);
    }
    agg.count.incrementAndGet(); // update count
    agg.sum.addAndGet(value);    // update sum
  }

  public double getAverageForKey(int key) {
    Aggregate agg = counts.get(key);
    return (agg != null && agg.count.get() != 0) ?
        (agg.sum.get()/((double)agg.count.get())) : 0;
  }
  public int getCountForKey(int key) { /* similar */ }
  public int getSumForKey(int key) {/* similar */ }
}
```

Map that contains an `Aggregate` object for each key.

`Aggregate` object contains count and value sum as atomic integer.

Is there a problem?
We did every thing right, didn't we?
We used `AtomicIntegers`, so we should not miss any update.

Test Program indicates a Problem

```
private static final int NUM_THREADS = 8;
private static final GroupCount groupCount = new GroupCount();

public static void main(String[] args) {
  Thread[] threads = new Thread[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++) {
    threads[i] = new Thread(() -> {
      for (int j = 0; j < 10000; j++) {
        groupCount.add(j % 10, 1);
      }
    });
    threads[i].start();
  }

  for (Thread t : threads) {
    try {
      t.join();
    } catch(InterruptedException e) { e.printStackTrace(); }
  }
  for (int i = 0; i < 10; i++) {
    System.out.println("group: " + i +
        ", count = " + groupCount.getSumForKey(i) +
        ", sum = " + groupCount.getCountForKey(i) +
        ", average = " + groupCount.getAverageForKey(i));
  }
}
```

Each thread adds 1,000 values for each of the 10 keys

Output:

```
group: 0, count = 7991, sum = 7991, average = 1.0
group: 1, count = 7995, sum = 7995, average = 1.0
group: 2, count = 7955, sum = 7955, average = 1.0
group: 3, count = 7995, sum = 7995, average = 1.0
group: 4, count = 7996, sum = 7996, average = 1.0
group: 5, count = 7996, sum = 7996, average = 1.0
group: 6, count = 7998, sum = 7998, average = 1.0
group: 7, count = 7999, sum = 7999, average = 1.0
group: 8, count = 7997, sum = 7997, average = 1.0
group: 9, count = 7997, sum = 7997, average = 1.0
```

- Since there are 8 threads, each group should have 8,000 values
- There are not 8,000 values/group
- However, **sum == count** for every group!
- If we are missing an update to **sum**, we are also missing it for **count**.

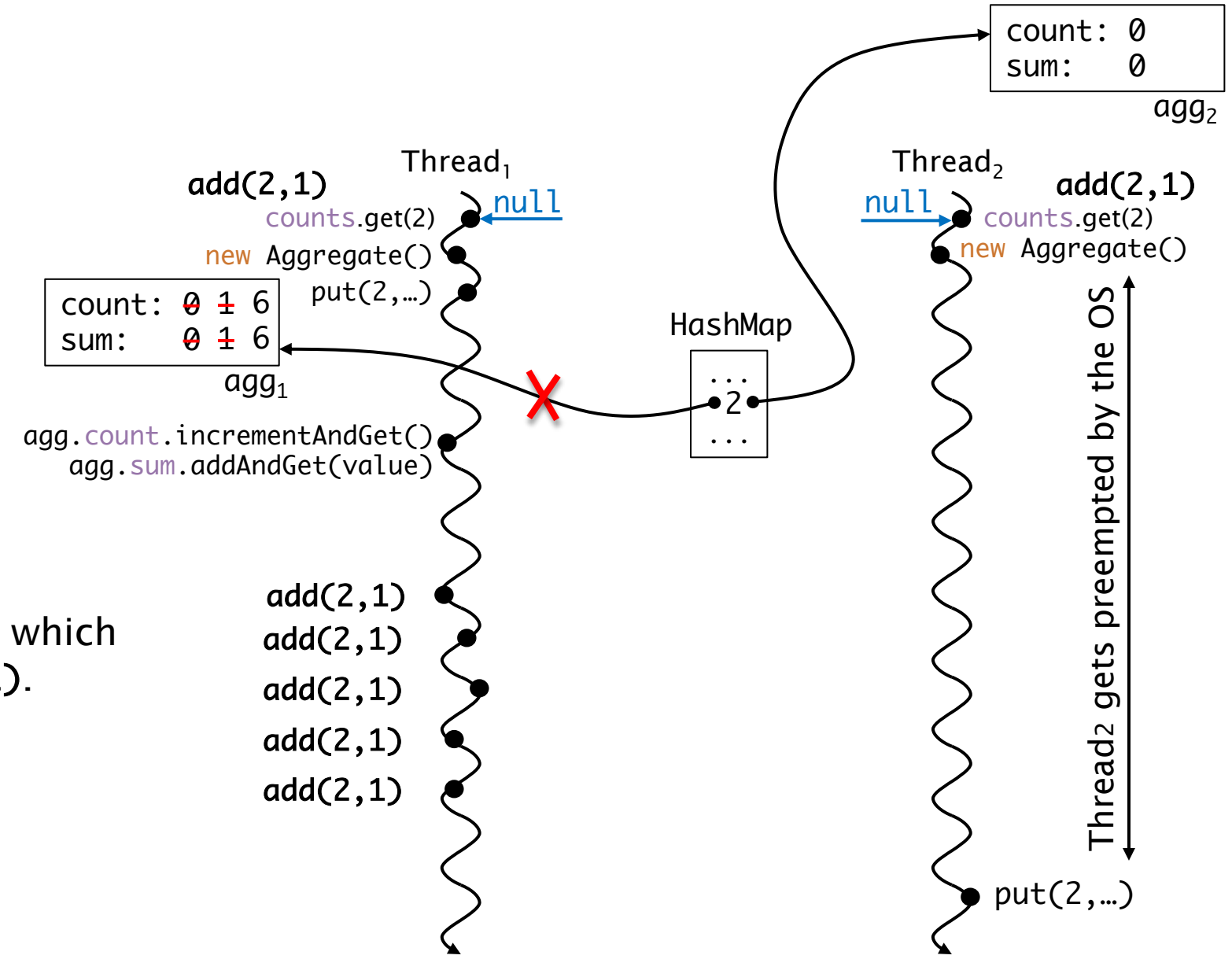
→ *What is going on?*

What went wrong?

- The add() method:

```
public void add(int key, int value) {
  Aggregate agg = counts.get(key);
  if (agg == null) {
    agg = new Aggregate();
    counts.put(key, agg);
  }
  agg.count.incrementAndGet();
  agg.sum.addAndGet(value);
}
```

- Consider the following scenario in which Thread$_1$ and Thread$_2$ call add(2, 1).



→ Updates of Thread$_1$ are lost!

Analysis of the Problem

- Using **AtomicInteger** alone is not sufficient to prevent Race Conditions.
- Multiple operations need to be formed atomically, not just the increment/add.

```
public void add(int key, int value) {
  Aggregate agg = counts.get(key);
  if (agg == null) {
    agg = new Aggregate();
    counts.put(key, agg);
  }
  agg.count.incrementAndGet();
  agg.sum.addAndGet(value);
}
```

- No thread is allowed to get between a `counts.get(key)` and a `counts.put(key, agg)`.
  → Statement sequence `get()` – `if` – `put()` must be atomic!

Critical Sections and Mutual Exclusion

```
public void add(int key, int value) {
  Aggregate agg = counts.get(key);
  if (agg == null) {
    agg = new Aggregate();
    counts.put(key, agg);
  }
  agg.count.incrementAndGet();
  agg.sum.addAndGet(value);
}
```

← Critical section, at most one thread may be inside it at any time!

- The code segment is called a Critical Section.
- At most one thread may be inside a critical section at any time.

- Threads must be mutually excluded from running the critical section.
  This is called Mutual Exclusion.
  - If a thread is already inside the critical section, a second thread that wants to enter, must wait until the first thread has left the section.
  - The 2nd thread blocks (the OS places it on a wait list).

Monitors

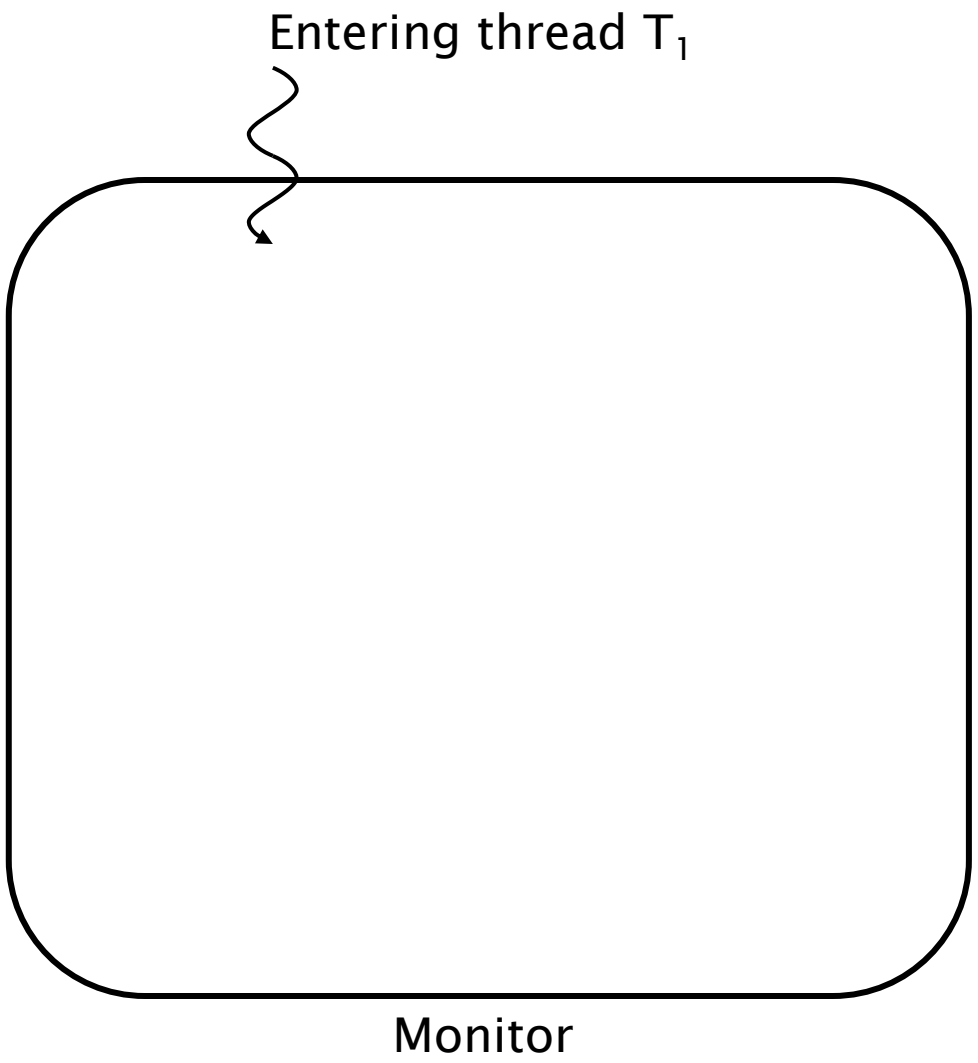
- In Java, Mutual Exclusion is implemented via Monitors.
- A monitor is an object whose methods can only be executed by at most one thread at any given time.
  - Monitors were invented by Per Brinch Hansen and C. A. R. Hoare in the 1970s.
  - Java does not quite implement Monitors correctly according to Brinch Hansen & Hoare though. The details are unfortunately beyond the scope of this class.

- Any Java object can serve as a monitor. To be used as a monitor it needs to be specified in the `synchronized` statement:

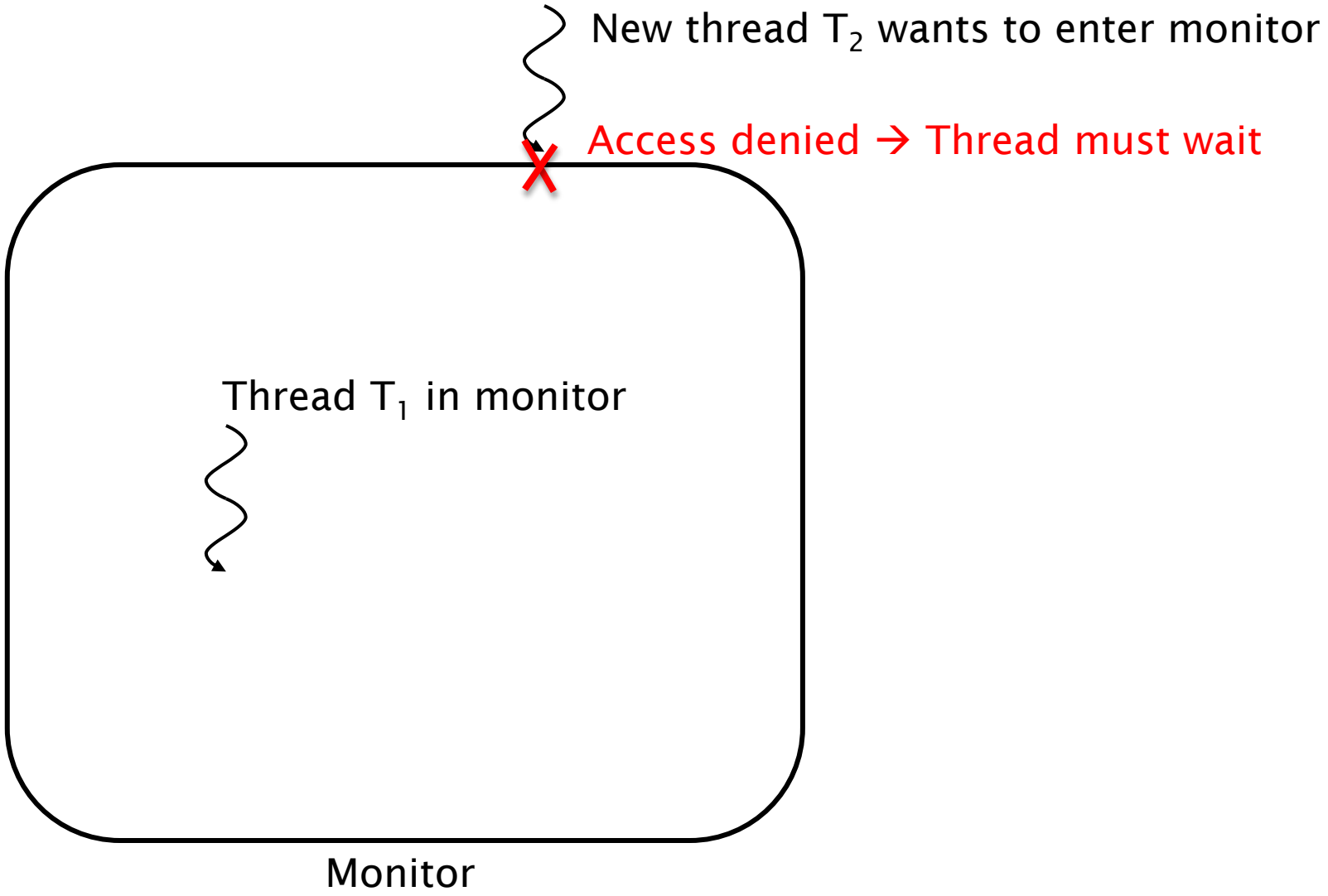
```
public void add(int key, int value) {
  Aggregate agg;
  synchronized (this) {
    agg = counts.get(key);
    if (agg == null) {
      agg = new Aggregate();
      counts.put(key, agg);
    }
  }
  agg.count.incrementAndGet();  // update count
  agg.sum.addAndGet(value);     // update sum
}
```

Monitor is the current object `this`.
Only one thread can be in **any code section** that is marked synchronized that uses **this** object as monitor.

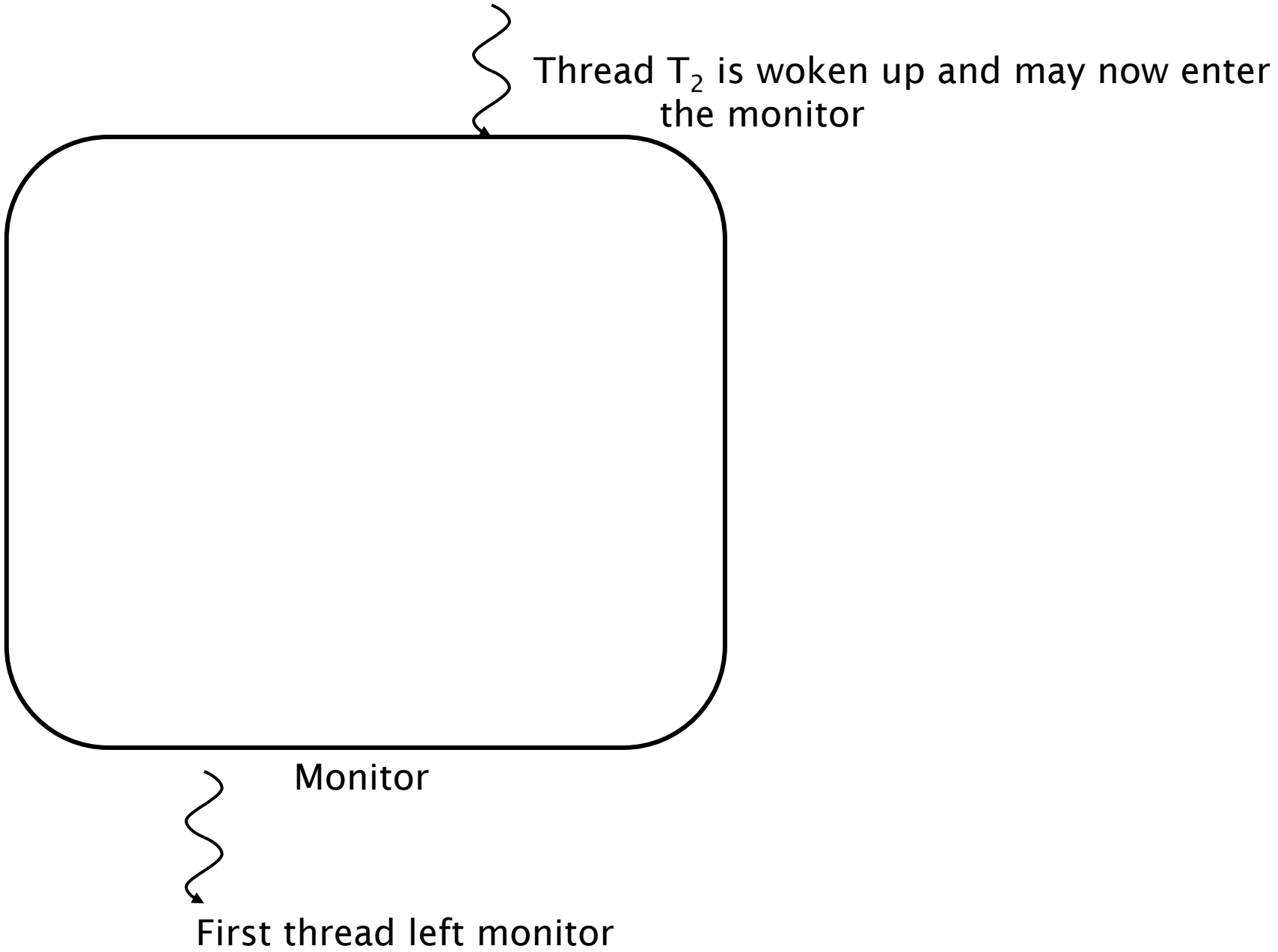
Monitors (2)



Monitors (3)


New thread T_2 wants to enter monitor
Access denied → Thread must wait
Thread T_1 in monitor
Monitor

Monitors (3)


Thread T_2 is woken up and may now enter the monitor
Monitor
First thread left monitor

Mutual Exclusion over Entire Method

Mutual exclusion can be extended to the full method. We can then replace the `AtomicInteger` for `sum` and `count` with simple `int`.

```
public void add(int key, int value) {
  synchronized (this) {
    Aggregate agg = counts.get(key);
    if (agg == null) {
      agg = new Aggregate();
      counts.put(key, agg);
    }
    agg.count++;            // update count
    agg.sum += value;       // update sum
  }
}
```

Synchronized Keyword in Method Declaration

Mutual exclusion can be extended to the entire method.

Java offers syntactic shortcut for that: Simple add the keyword synchronized to the method declaration.

```
public void add(int key, int value) {
  synchronized (this) {
    Aggregate agg = counts.get(key);
    if (agg == null) {
      agg = new Aggregate();
      counts.put(key, agg);
    }
    agg.count++;
    agg.sum += value;
  }
}
```

==

```
public synchronized void add(int key, int value) {
  Aggregate agg = counts.get(key);
  if (agg == null) {
    agg = new Aggregate();
    counts.put(key, agg);
  }
  agg.count++;
  agg.sum += value;
}
```

Rerunning Test Program

```
private static final int NUM_THREADS = 8;
private static final GroupCount groupCount = new GroupCount();

public static void main(String[] args) {
  Thread[] threads = new Thread[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++) {
    threads[i] = new Thread(() -> {
      for (int j = 0; j < 10000; j++) {
        groupCount.add(j % 10, 1);
      }
    });
    threads[i].start();
  }

  for (Thread t : threads) {
    try {
      t.join();
    } catch(InterruptedException e) { e.printStackTrace(); }
  }
  for (int i = 0; i < 10; i++) {
    System.out.println("group: " + i +
        ", count = " + groupCount.getSumForKey(i) +
        ", sum = " + groupCount.getCountForKey(i) +
        ", average = " + groupCount.getAverageForKey(i));
  }
}
```

Each thread adds 1,000 values for each of the 10 keys

Output:

```
group: 0, count = 8000, sum = 8000, average = 1.0
group: 1, count = 8000, sum = 8000, average = 1.0
group: 2, count = 8000, sum = 8000, average = 1.0
group: 3, count = 8000, sum = 8000, average = 1.0
group: 4, count = 8000, sum = 8000, average = 1.0
group: 5, count = 8000, sum = 8000, average = 1.0
group: 6, count = 8000, sum = 8000, average = 1.0
group: 7, count = 8000, sum = 8000, average = 1.0
group: 8, count = 8000, sum = 8000, average = 1.0
group: 9, count = 8000, sum = 8000, average = 1.0
```

All Good! ☺

Really…? Let's have the Threads check the Averages

```
private static final int NUM_THREADS = 8;
private static final GroupCountSync groupCount = new GroupCountSync();

public static void main(String[] args) {
  Thread[] threads = new Thread[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++) {
    threads[i] = new Thread(() -> {
      for (int j = 0; j < 10000; j++) {
        groupCount.add(j % 10, 1);
        double avg = groupCount.getAverageForKey(j % 10);
        if (Math.abs(avg-1.0) > 1.0e-6) {
          System.err.println("average not 1, got " + avg);
        }
      }
    });
    threads[i].start();
  }
  for (Thread t : threads) {
    try { t.join(); } catch(InterruptedException e) { e.printStackTrace(); }
  }
}
```

Output:

```
average not 1, got 1.000133904659882
average not 1, got 1.0001333511134818
average not 1, got 1.0002669514148426
average not 1, got 1.000263400500461
average not 1, got 1.00065711657248
average not 1, got 1.0001316482359137
average not 1, got 1.0002567394094994
...
```

→ Oops?

→ What went wrong this time?

Download `GroupCountSync.java` from Moodle

Last Fix for Group Count

- Since **getAverageForKey()** accesses both sum and count it must be in a critical section too.
  → Add synchronized keyword to **getAverageForKey()**.

Add() and **getAverageForKey()** are synchronized on the same monitor object (**this**), i.e., only one thread can be in either of the methods at any time.

```
public synchronized double getAverageForKey(int key) {
  Aggregate agg = counts.get(key);
  return (agg != null && agg.count != 0) ?
      (agg.sum/((double)agg.count)) : 0;
}
```

Performance of Counter

- Three version of the initial counter code:
  1. Using `int counter` → gives incorrect results due to data race
  2. Using `AtomicInteger counter` → gives correct result through atomic operation
  3. Using `int counter` + `synchronized` → gives correct result through mutual exclusion (monitor)

```
// Version 3: synchronized counter
counter = 0;
long start = System.nanoTime();
Thread[] threads = new Thread[NUM_THREADS];
for (int i = 0; i < threads.length; ++i) {
  threads[i] = new Thread(() -> {
    for (int j = 0; j < PER_THREAD_COUNT; j++) {
      synchronized (DataRace.class) { // synchronize on class object (no 'this' here)
        counter++;
      }
    }
  });
  threads[i].start();
}
```

Download `DataRaceBenchmark.java` from Moodle

Performance of Counter (2)

- MacBook Pro 15" (2017) i7-7920HQ, 4 cores
- Configuration:

```
private static final int NUM_THREADS = 8;
private static final int PER_THREAD_COUNT = 10000000;
```

Incorrect result but by far the fasted variant!

```
              counter = 20778564  expected 80000000 (8 threads, 0.085512107 s)
       atomic counter = 80000000  expected 80000000 (8 threads, 1.4524806330000002 secs)
synchronized counter = 80000000  expected 80000000 (8 threads, 3.2876580580000003 secs)
```

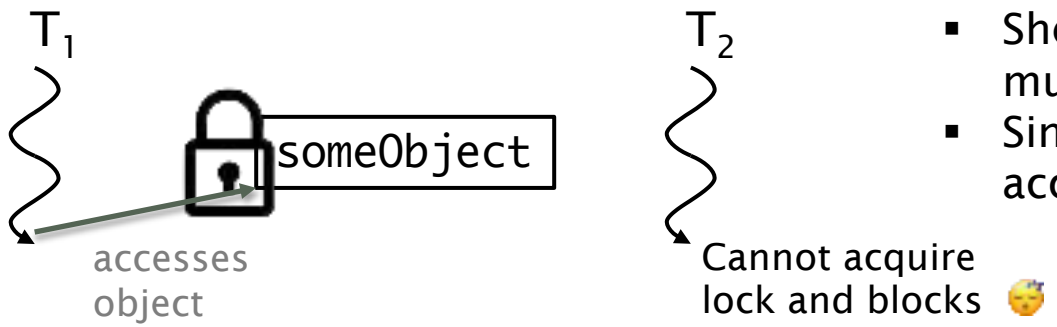
Correct result but very slow!

Observation:

- Atomic Operation is significantly slower than `int` additions.
  - Atomic Operations are expensive.
    8 threads are hammering on a single 32-bit word in memory → corresponding cache line bounces between the (non-shared) L1 caches of the 4 CPU cores running the 8 threads.
- Monitor approach is even slower.
  - Monitors require threads to be suspended and woken up → takes even more time a cache line bouncing between cores.

Mutual Exclusion using Locks

- Locks (also called Mutexes and Latches) allow a fine-granular control of mutual exclusion.
- Lock are used to control access to shared resources and data structures (e.g., `counter`).
- Threads that want to access the resource or data structure need to obtain the lock first.
- Rule: Only one thread can hold a lock at any time.
  - If a thread T_2 tries to acquire a lock that is already held by thread T_1, T_2 must wait until T_1 releases it. T_2 blocks (sleeps).
  - Once T_1 releases the lock, T_2 woken up and successfully acquires the lock unless T_2 looses out against some other thread T_3 that also competes for the lock.



- Programmer chooses to protect `someObject` with a lock.
- She demands that anyone thread that accesses the object must hold the lock.
- Since only one thread can hold the lock, only one thread can access the object at any time.

Locks in Java

- In Java, Locks are provided by the class `ReentrantLock` in the package `java.util.concurrent.locks`.
- Locks have the following methods (from the Interface `Lock`):
  - `void lock();` calling thread acquires lock, will have to wait if it cannot get lock
  - `void unlock();` calling thread releases lock (it must currently hold it)
  - `boolean tryLock();` calling tries to acquire lock, returns `false` if it cannot get lock, `true` otherwise. Call will **not block**.

- A lock is **reentrant** if the same thread that is already holding the lock can reacquire the same lock without blocking.

Counter Example using Locks

```
public class LockTest {

  private static final int NUM_THREADS = 8;
  private static final int PER_THREAD_COUNT = 100000;
  private static int counter = 0;

  public static void main(String[] args) {
    Lock lock = new ReentrantLock();

    Thread[] threads = new Thread[NUM_THREADS];
    for (int i = 0; i < threads.length; ++i) {
      threads[i] = new Thread(() -> {
        for (int j = 0; j < PER_THREAD_COUNT; j++) {
          lock.lock();
          counter++;  // Critical Section
          lock.unlock();
        }
      });
      threads[i].start();
    }
    for (Thread t : threads) {
      try { t.join(); } catch (InterruptedException e) { e.printStackTrace(); }
    }
    System.out.println("counter = "+counter+"  expected " + (PER_THREAD_COUNT * NUM_THREADS) +
        " ("+NUM_THREADS+" threads)");
  }
}
```

Thread tries to acquire the lock, blocks until it becomes available.

Thread releases lock, if there is one thread waiting for the lock it gets woken up.

Correct output:

```
counter = 800000  expected 800000 (8 threads)
```

Using Locks vs. using synchronized

- A critical section demarked using `synchronized` either spans a single method when used in the method declaration or a block of statements when used inside a method.
  - Compiler automatically generates code to exit the monitor whenever the control flow leaves the synchronized block or method.
- Locks demark critical section between the `lock()` and `unlock()`.
- Locks, on the other hand must be explicitly released by calling `unlock()`. If you forget to release a lock the program will subsequently hang.

- Locks can be acquired and released within an individual methods
  - Then locks do not offer any additional advantages over the use of `synchronized`.
  - In fact, they are very often lead to bugs, e.g., the method terminates by throwing an exception and the programmer forgot to call `unlock()` before throwing the exception. The lock is not released and subsequent `lock()` will cause the program to hang.

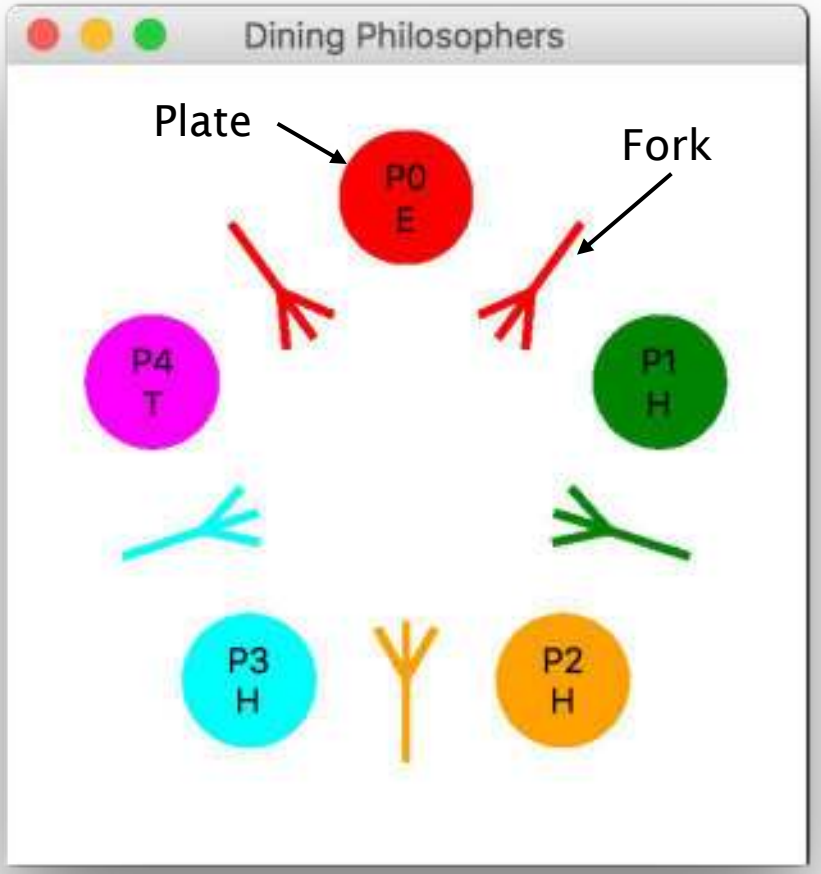
- Locks can be acquired explicitly and locks can be held for multiple method invocations.

Deadlocks

Dining Philosophers

- 5 silent philosopher sitting around a table
- There is one fork between every neighboring philosophers
- The 5 philosophers forever do
  1. Think for a random amount of time
  2. Get hungry.
  3. Eat for a random amount of time until they are full.
- Catch: A philosopher always needs his left and right hand fork in order to eat.

- **Question**: How can they eat without starving?



Legend
P0 – P4: Philosophers
T: "Thinking", H: "Hungry", E: "Eating"

Implementation in Java FX

Download `DiningPhilosophers.java` from Moodle.

```
for (int i = 0; i < 5; i++) {    // Start philosophers' work threads
  final int pid = i;             // philosopher's ID
  Thread thread = new Thread(() -> {
    final Random rnd = new Random();

    while (!Thread.interrupted()) { // repeat until interrupted
      try { // Philosopher thinks for some random time.
        Thread.sleep(rnd.nextInt(MAX_THINK_TIME_MS));
      } catch(InterruptedException ignored) { return; }

      // Philosopher is hungry and wants to eat.
      philosopherState[pid] = HUNGRY;
      takeLeftFork(pid);
      takeRightFork(pid);
      philosopherState[pid] = EATING;

      try { // Philosopher eats for some random time.
        Thread.sleep(rnd.nextInt(MAX_EAT_TIME_MS));
      } catch(InterruptedException ignored) {
        releaseLeftFork(pid); releaseRightFork(pid); // Release both forks when interrupted.
        return;
      }
      philosopherState[pid] = THINKING;
      releaseLeftFork(pid);
      releaseRightFork(pid);
    }
  });
  thread.setDaemon(true);
  thread.start();
}
```

Philosopher thinks for some time.
Here, we let him sleep for a random time.

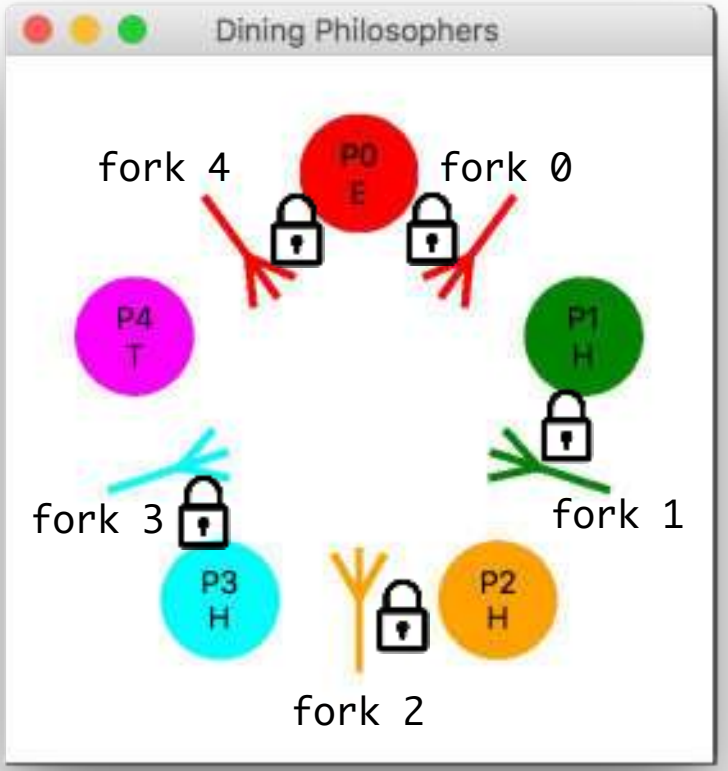
Hungry philosopher wants to eat.
He takes first his left fork then his right-hand fork.
Once he has both forks he starts eating

We, het the philosopher eat for some random time (modelled as sleeping).

Philosopher releases left and right Fork and resumes thinking.

Forks are Protected by Locks

- Each fork is protected by a lock.
- A philosopher thread must acquire both locks before it can start eating.
- **Description of Figure**
  - Philosopher 0 has left (0) and right (4) fork's lock and is eating.
  - Philosopher is hungry, has acquired the lock of its left fork (1) but must wait until it can also get the right fork's lock (0).
  - Philosopher 2 is hungry, has acquired the lock of its left fork (2) but must wait until it can also get the right fork's lock (1).
  - Philosopher 3 is hungry, has acquired the lock of its left fork (3) but must wait until it can also get the right fork's lock (2).
  - Philosopher 4 is thinking.



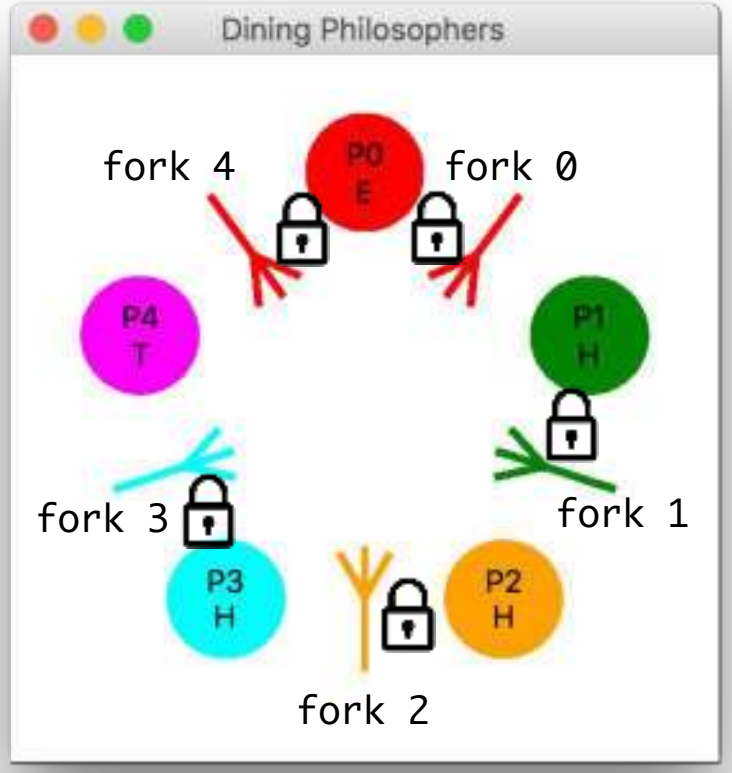
Implementation in Java FX (2)

Download `DiningPhilosophers.java` from Moodle.

```
/** State of the each philosopher. */
private PhilosopherState[] philosopherState = {THINKING, THINKING, THINKING, THINKING, THINKING};

/** entry[i] == j, indicates that the left-hand side fork of philosopher i is used by philosopher j,
 * entry[i] == -1, indicates that the left-hand side fork of philosopher i is currently not used. */
private int forkUsedByPhilosopher[] = new int[philosopherState.length];

/** Java Locks for each fork. */
private ReentrantLock[] forks = new ReentrantLock[philosopherState.length];

private void takeLeftFork(int philosopherId) {
  forks[philosopherId].lock();
  forkUsedByPhilosopher[philosopherId] = philosopherId;
}

private void takeRightFork(int philosopherId) {
  int idx = (philosopherId + forks.length - 1) % forks.length;
  forks[idx].lock();
  forkUsedByPhilosopher[idx] = philosopherId;
}

private void releaseLeftFork(int philosopherId) {
  forkUsedByPhilosopher[philosopherId] = -1;
  forks[philosopherId].unlock();
}

private void releaseRightFork(int philosopherId) {
  int idx = (philosopherId + forks.length - 1) % forks.length;
  forkUsedByPhilosopher[idx] = -1;
  forks[idx].unlock();
}
```

Every fork is protected by a lock.

A philosopher thread must **acquire both locks** before it can start eating.

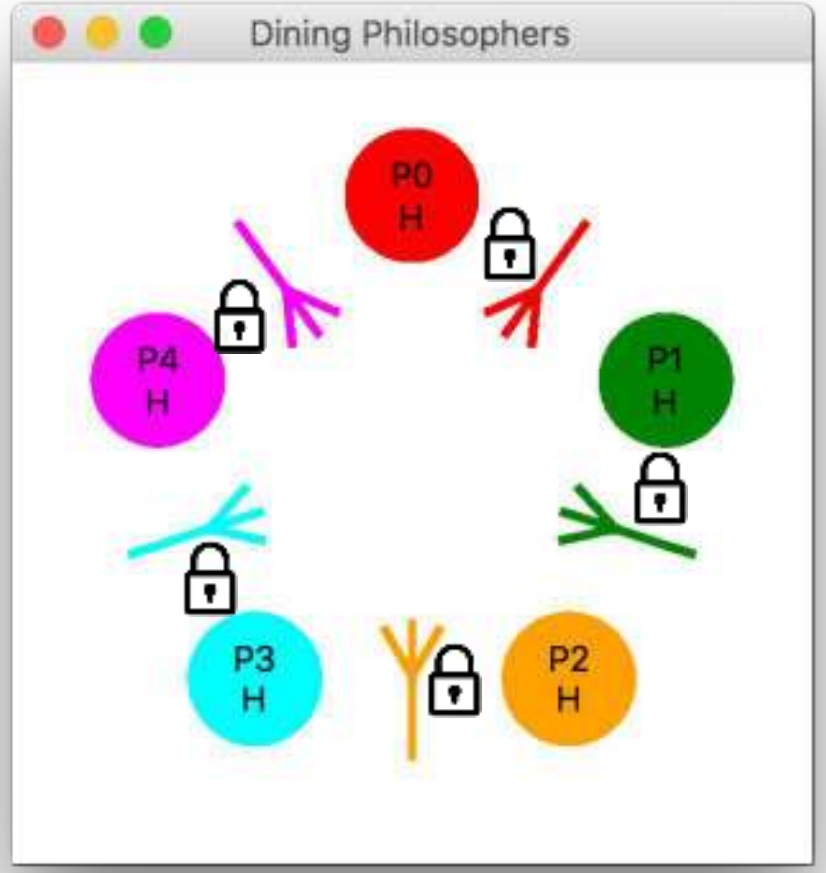


What will eventually happen… a Deadlock

- The philosophers will all collectively starve!
- Each philosopher has acquired its left fork and waits for the second fork.
  - Nobody gets the second fork.
  - Nobody gets to eat.
  - Everybody starves.

- This situation is called a Deadlock.

  1. Every philosopher is hungry.
  2. Every philosopher has acquired his left fork's lock.
  3. No philosopher can get right fork's lock because it is currently held by another philosopher.
  4. All philosopher threads are blocked.
  5. No thread can continue and make progress.
  6. Simulation is stuck!



Deadlock

- Deadlocks occur in multithreaded programs that use shared resources.
- Deadlock: a situation in which threads that want to access a shared resource are forced to wait because the resource is held by another thread, which in turn is waiting for a resource that is held by a waiting thread.

- Necessary conditions for Deadlock Situation (**all** must be satisfied simultaneously):
  1. Resources are held exclusively by threads (mutual exclusion).
  2. A thread that currently already holds a resource can request additional resources.
  3. Resources cannot be forcibly taken away (no preemption).
  4. Circular wait condition. Thread T_1 is waiting for resource held by T_2, T_2 is waiting for resource held by T_3, …, Thread T_n is waiting for resource held by T_1.

Starving Philosophers – Deadlock Analysis

- Necessary condition for Deadlock Situation (all must be satisfied simultaneously):
  - Resources are held exclusively by threads (mutual exclusion).

  A fork can only be held by at most one thread.

  - A thread that currently already holds a resource can request additional resources

  After a philosopher has successfully acquired its left fork, he requests the right fork.

  - Resources cannot be forcibly taken away (no preemption).

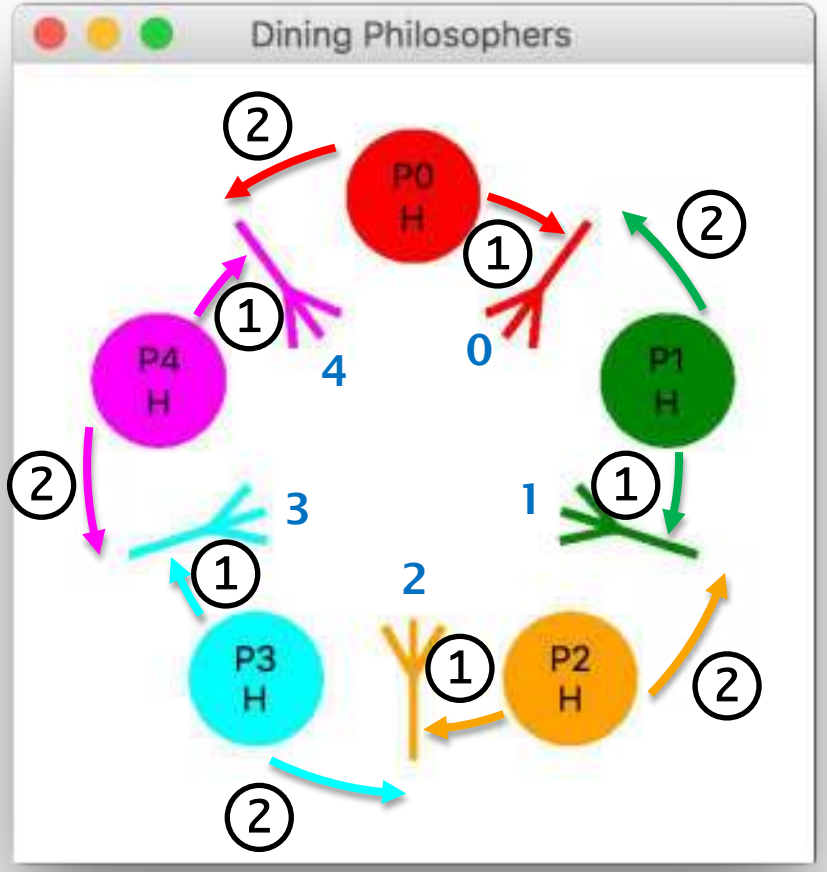
  Once a philosopher has acquired a fork it, he only releases it when he is done eating. Nobody takes him the fork away forcefully.

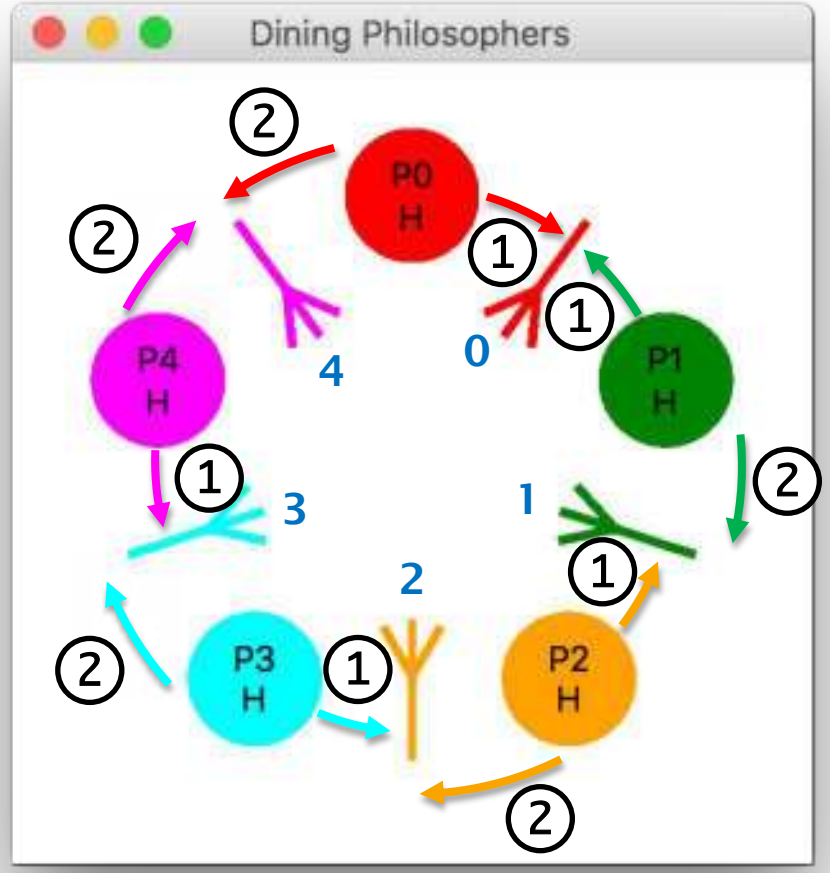
  - Circular wait condition. Thread T_1 is waiting for resource held by T_2, T_2 is waiting for resource held by T_3, …, Thread T_n is waiting for resource held by T_1.

  Circular wait condition is possible (see picture on previous slide).

- Deadlock Prevention: Prevent one of the four conditions from occurring!

Preventing the Circular Wait Condition





Deadlock: Circular Wait

1. Inner circle: `takeLeftFork(pid);`
2. Outer circle: `takeRightFork(pid);`

No Deadlock: No Circular Wait

1. Take fork with smaller index first
2. Take fork with larger index second

Deadlock-free Philosophers (1)

Download `DiningPhilosophersDeadlockFree.java` from Moodle.

```
for (int i = 0; i < 5; i++) { // Start philosophers' work threads
  final int pid = i;          // philosopher's ID
  Thread thread = new Thread(() -> {
    final Random rnd = new Random();
    int firstFork = Math.min(pid, (pid + 5 - 1) % 5);
    int secondFork = Math.max(pid, (pid + 5 - 1) % 5);

    while (!Thread.interrupted()) { // Repeat until interrupted.
      try { // Philosopher thinks for some random time.
        Thread.sleep(rnd.nextInt(MAX_THINK_TIME_MS));
      } catch(InterruptedException ignored) { return; }

      // Philosopher is hungry and wants to eat.
      philosopherState[pid] = HUNGRY;
      takeFork(firstFork, pid);
      takeFork(secondFork, pid);
      philosopherState[pid] = EATING;

      try { // Philosopher eats for some random time.
        Thread.sleep(rnd.nextInt(MAX_EAT_TIME_MS));
      } catch(InterruptedException ignored) { // Release both forks when interrupted.
        releaseFork(firstFork); releaseFork(secondFork); return; }
      philosopherState[pid] = THINKING;
      releaseFork(firstFork);
      releaseFork(secondFork);
    }
  });
  thread.setDaemon(true);
  thread.start();
}
```

The first fork to acquire is the fork with the lower number.
The second fork to acquire is the fork with the higher number.

Acquire forks

Deadlock-free Philosophers (2)

```
/** State of the each philosopher. */
private PhilosopherState[] philosopherState = {THINKING, THINKING, THINKING, THINKING, THINKING};

/** entry[i] == j, indicates that the left-hand side fork of philosopher i is used by philosopher j,
 * entry[i] == -1, indicates that the left-hand side fork of philosopher i is currently not used. */
private int forkUsedByPhilosopher[] = new int[philosopherState.length];

/** Java Locks for each fork. */
private ReentrantLock[] forks = new ReentrantLock[philosopherState.length];

private void takeFork(int forkId, int philosopherId) {
  forks[forkId].lock();
  forkUsedByPhilosopher[forkId] = philosopherId;
}

private void releaseFork(int forkId) {
  forkUsedByPhilosopher[forkId] = -1;
  forks[forkId].unlock();
}
```

Exercise 4: Problem 1

Concurrent Bank that is safe (consistent, without race-conditions and deadlocks)

Synchronization of Threads

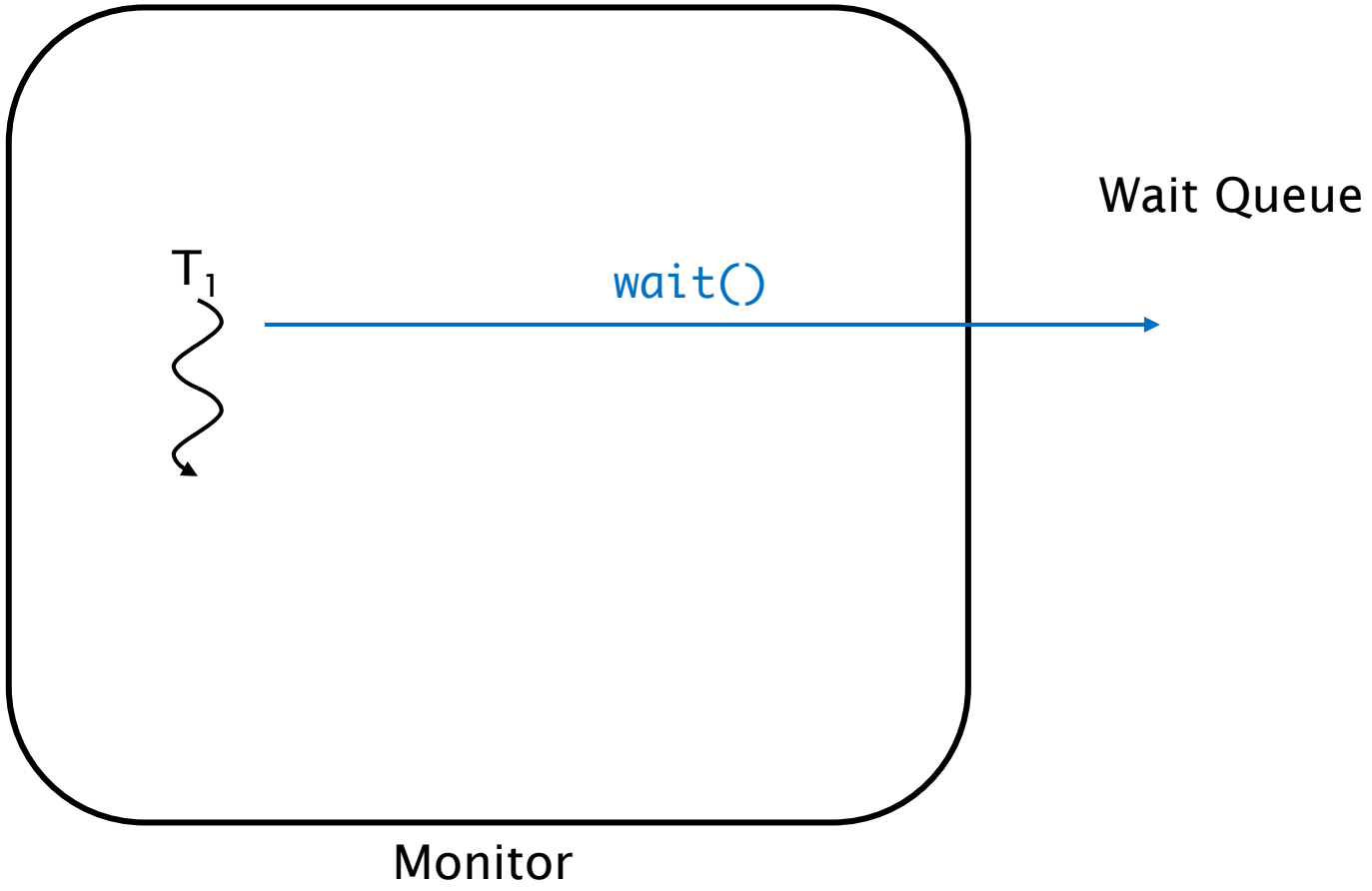
Synchronization between Thread

- Threads often have to collaborate. Collaboration requires synchronization.
- A thread T_1 may have to wait for the result produced by another thread T_2 before it can proceed.
  - T_1 cannot proceed. T_1 waits for T_2. → T_1 blocks.
  - Once T_2 has the result ready, T_2 notifies or wakes up thread T_1.

- Specifically:
  1. T_1 is in a critical section (a monitor) and notices that it cannot proceed because the data produced by T_2 is not ready.
  2. T_1 calls `wait()`. → It leaves the monitor temporarily and is put on the scheduler's *wait queue*.
  3. After T_2 has computed the data, it enters the monitor. T_2 signals (notifies, wakes up) T_1. T_1 is put from the *wait queue* onto the *ready queue*. T_1 now competes with other threads to enter the monitor.
  4. T_2 leaves the monitor after notification.
  5. T_1 can enter the monitor and resume its computation.

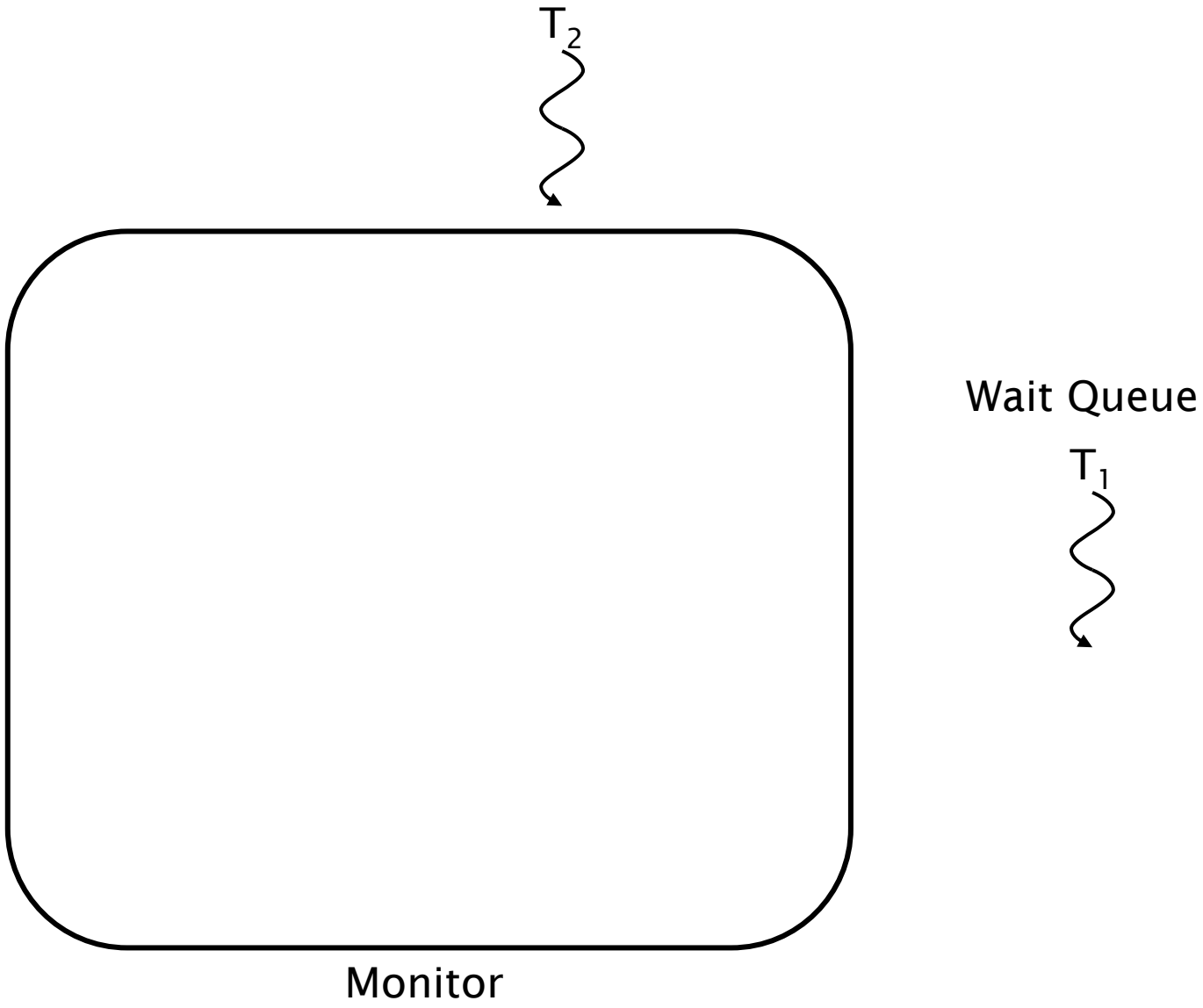
wait(), notify() (1)

- T_1 is in the monitor.
- T_1 needs to wait for result computed by a thread T_2.
- T_1 calls `wait()`.



wait(), notify() (2)

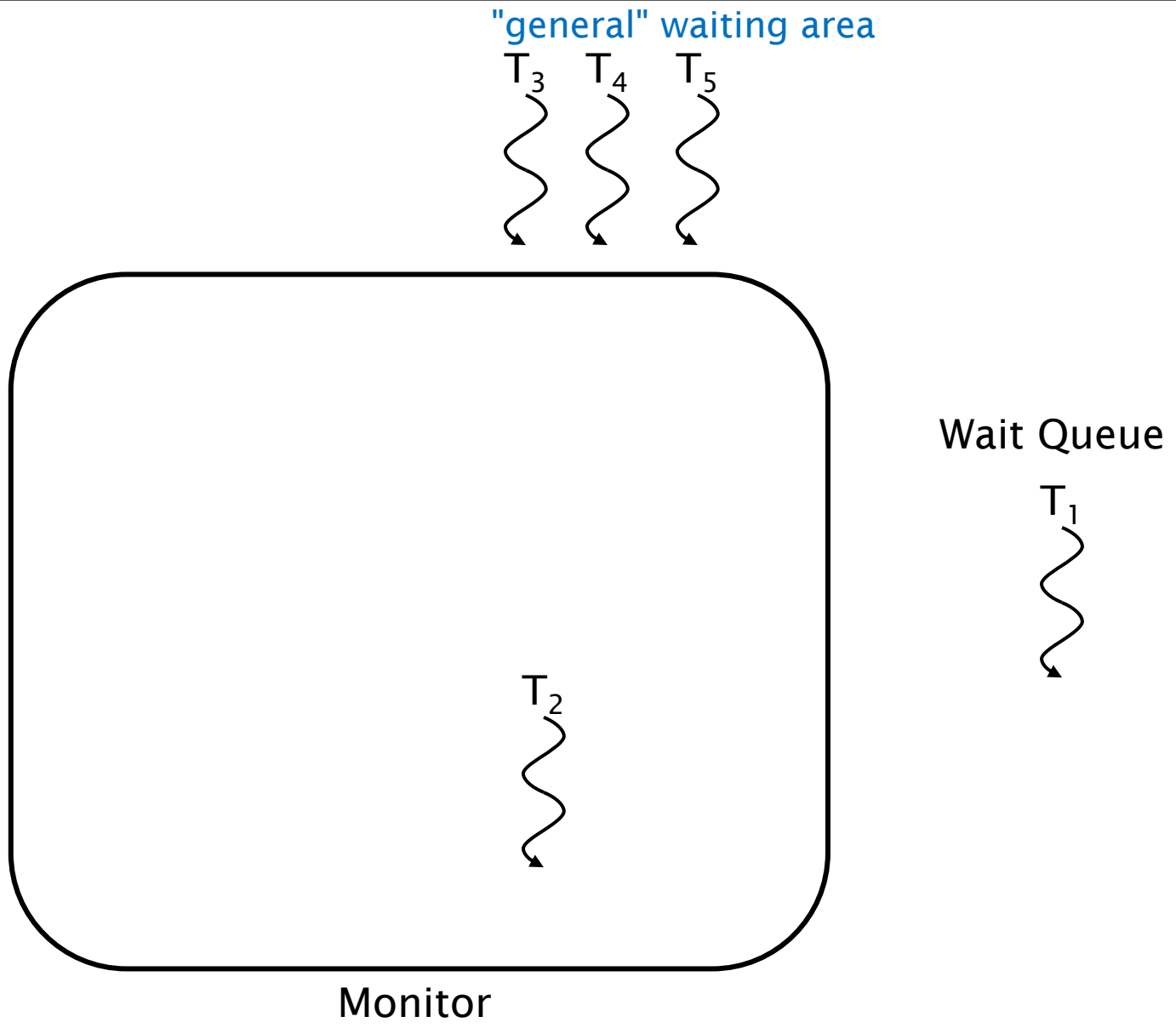
- T_2 enters the monitor



wait(), notify() (3)

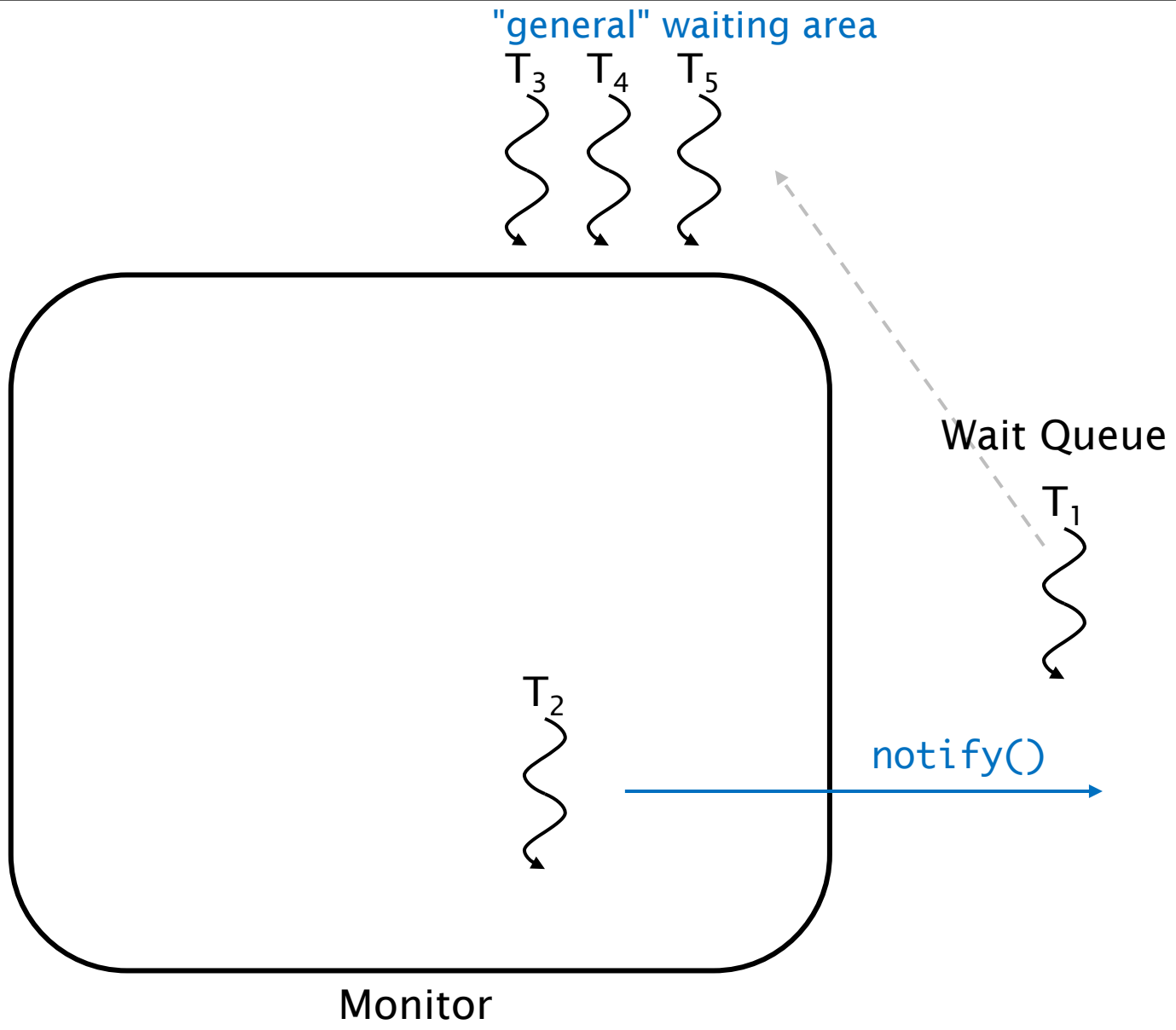
- T_3, T_4, T_5 want to enter the monitor.
- They cannot enter since monitor is occupied by T_2.
- They need to wait in the "general" waiting area.

- Once T_2 leaves the monitor, one of them (non-deterministic) is allowed to enter.



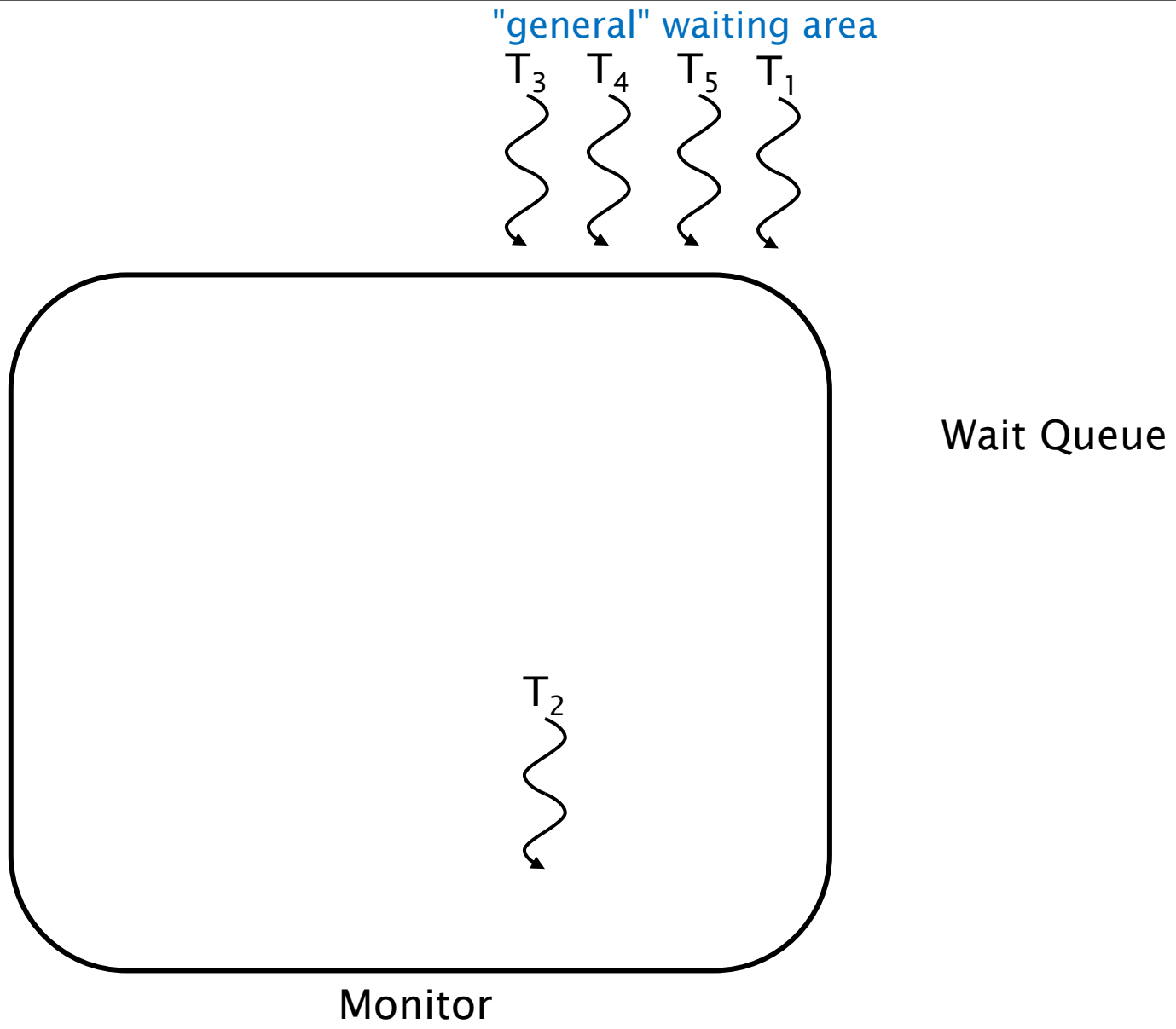
wait(), notify() (4)

- T_2 has produced the data T_1 is waiting for.
- T_2 calls `notify()`, which wakes up at one thread on the waiting queue.
- T_1 cannot enter monitor yet because T_2 is still inside it.
- T_2 needs to wait with all other threads in the "general" waiting area to reenter.



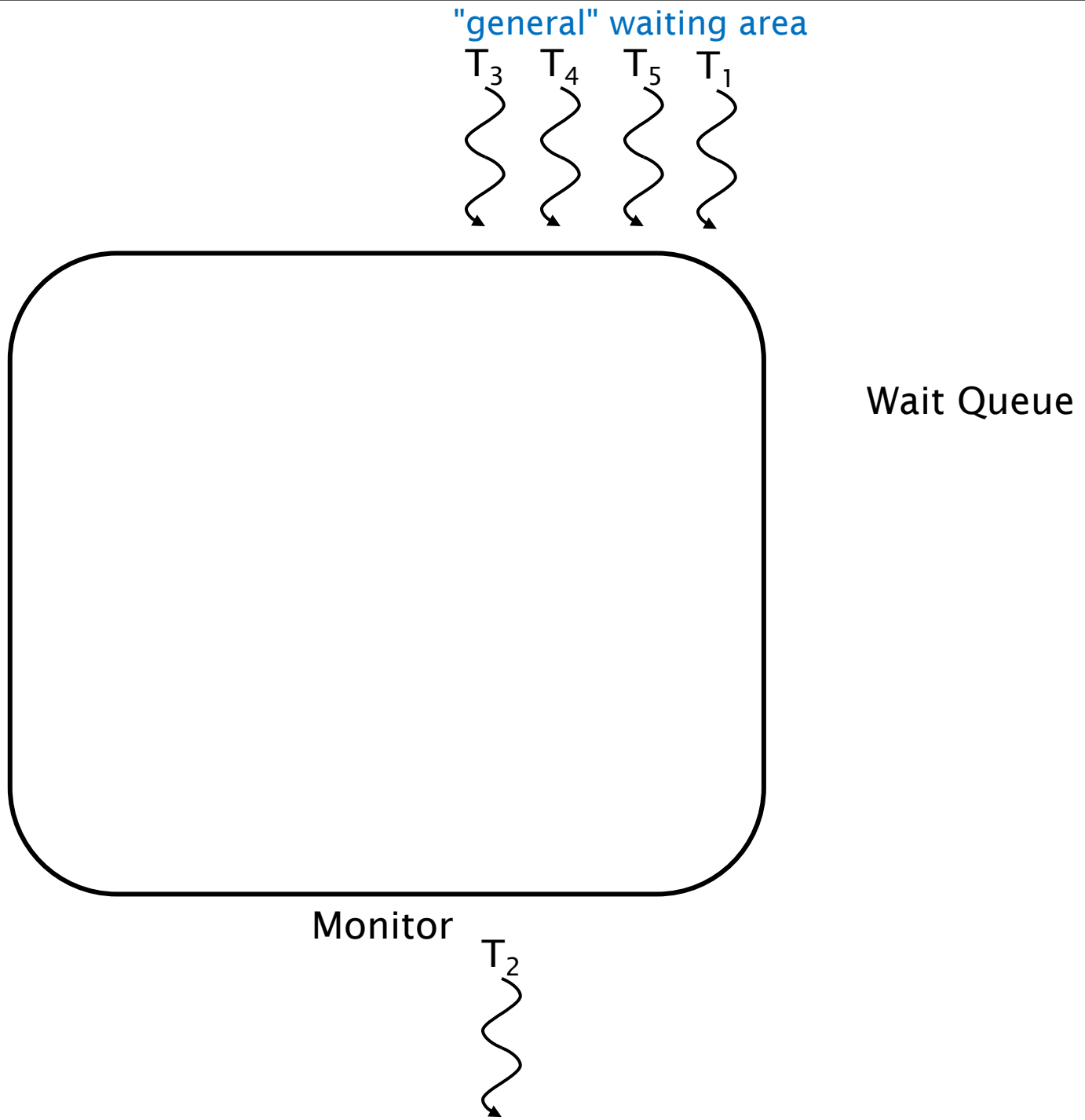
wait(), notify() (5)

- T_2 leaves monitor.



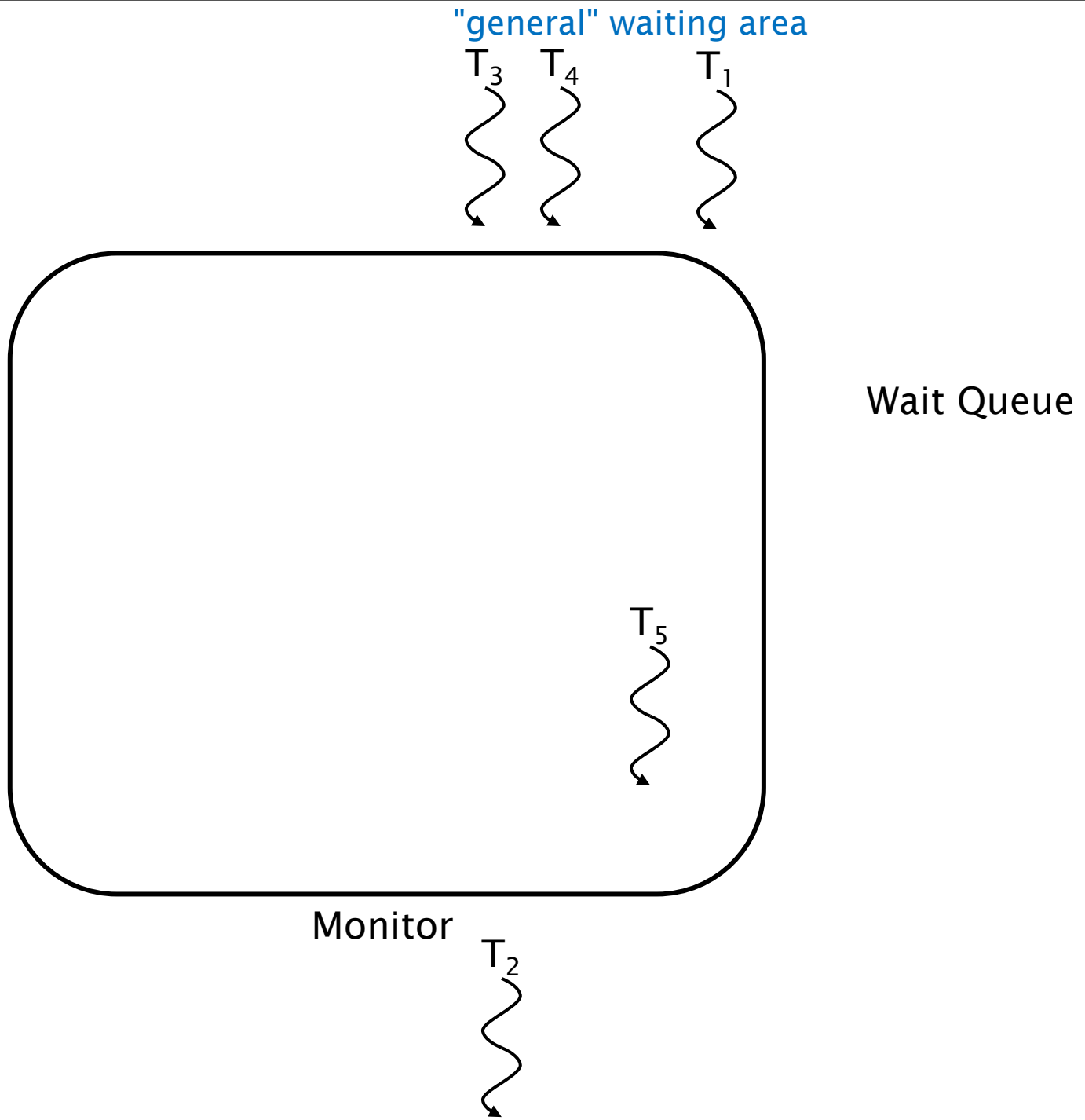
wait(), notify() (6)

- T_5 wins the competition.
- T_5 enters the monitor.



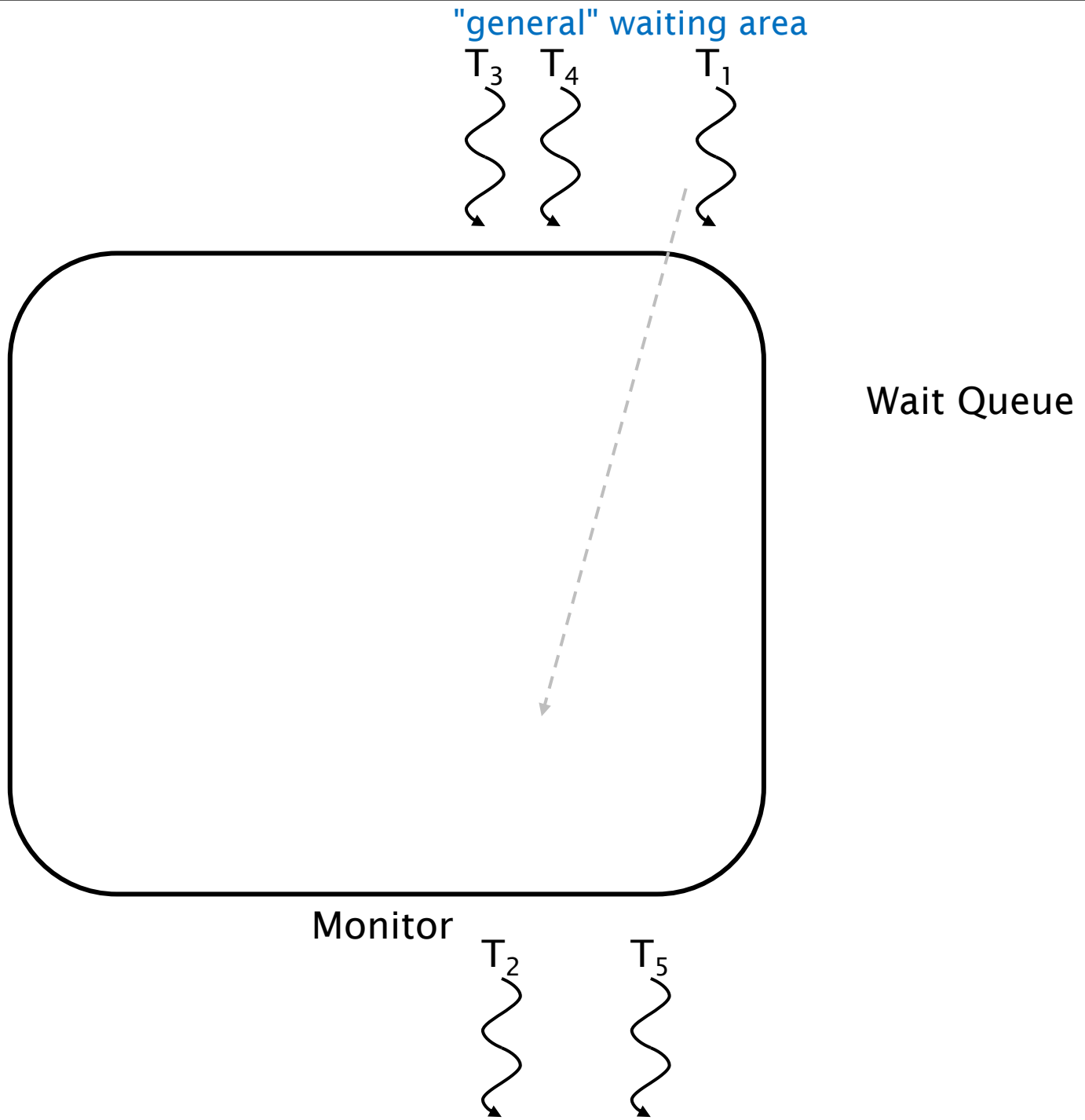
wait(), notify() (7)

- T_5 leaves the monitor.



wait(), notify() (8)

- T_1 wins the competition.
- T_1 finally can enter the monitor and continue with its computation.

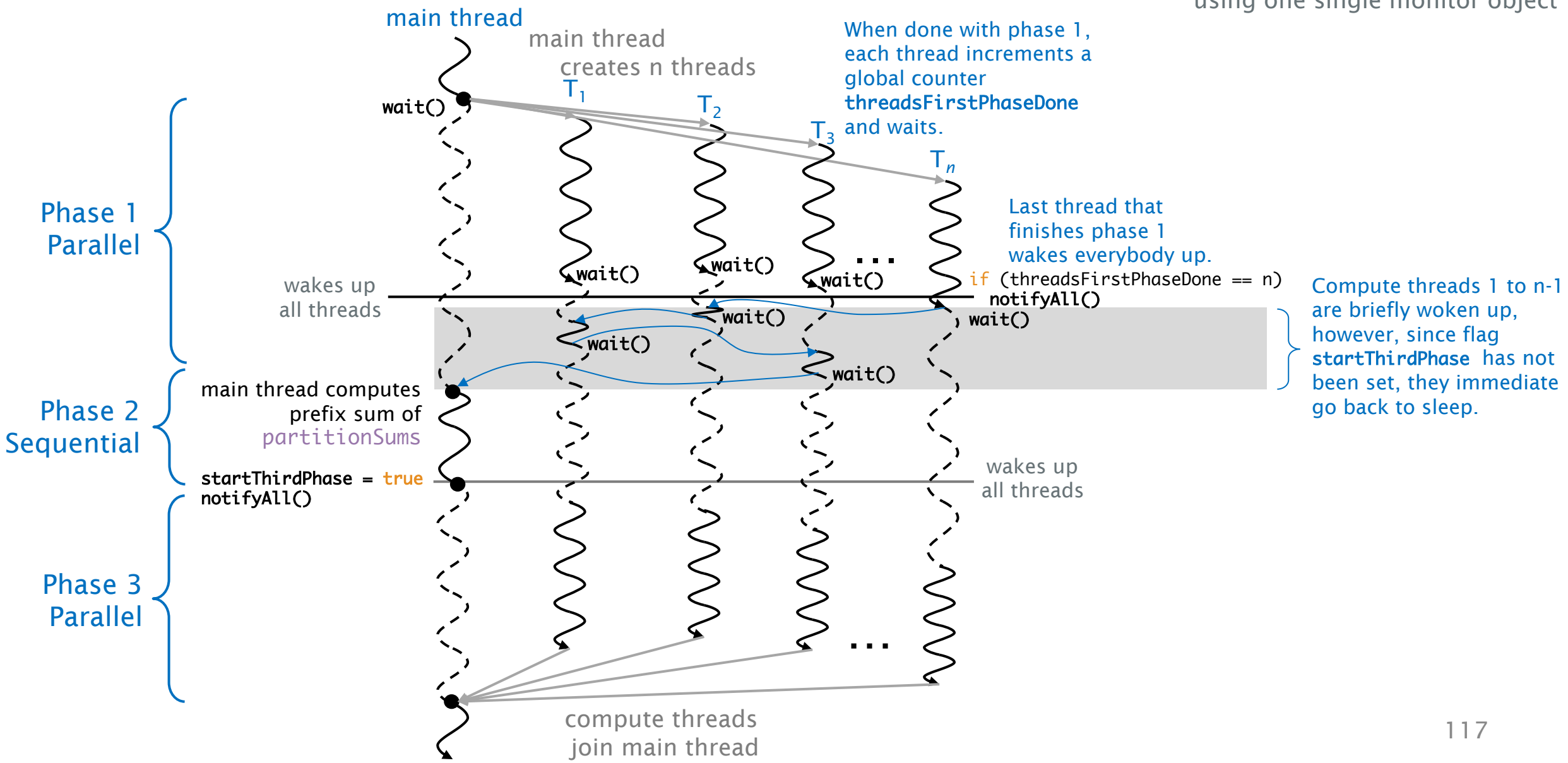


wait(), notify(), and notifyAll() Methods of Class Object

- The class `Object` provides the following method for synchronization on an object's monitor.
  - `public final void wait() throws InterruptedException`
    The object currently in the monitor exits the monitor and waits until it gets notified by another thread.
    The thread that calls this method must be inside the monitor (in `synchronized` block or method) otherwise an `IllegalMonitorStateException` is thrown.
  - `public final void notify()`
    Thread currently in the monitor notifies one thread that is waiting on the object's monitor. The awakened thread may not enter the monitor until the current thread leaves it. The awakened thread competes with all other threads that wish to enter the monitor.
  - `public final void notifyAll()`
    Thread currently in the monitor notifies all threads that are waiting on the object's monitor. None of the awakened threads may not enter the monitor until the current thread leaves it. The awakened threads compete with all other threads that wish to enter the monitor.

Example: Parallel Prefix Sum with Synchronization

using one single monitor object



Parallel Prefix Sum using Single Monitor (1)

```
private static void parallelPrefixSumWaitGenRandom() {
  Thread[] threads = new Thread[NUM_THREADS];
  final int elementsPerThread = LENGTH / NUM_THREADS;
  threadsFirstPhaseDone = 0;  // Counter variable incremented by each worker when phase 1 complete.
  startThirdPhase = false;    // Flag variable set by main thread to start phase 3.

  for (int i = 0; i < NUM_THREADS; i++) {
    final int tid = i
    threads[i] = new Thread(() -> {
      int myStartIdx = tid * elementsPerThread;      // determine start index for thread tid
      Random random = new Random();

      // Phase 1 (PARALLEL): prefix sum over partitions
      long mySum = 0;
      for (int j = 0; j < elementsPerThread; j++) {  // iterate over thread's partition of the data
        int newValue = random.nextInt();             // generate new random number
        mySum += newValue;                           // update cumulative sum for this thread
        input[myStartIdx + j] = newValue;
        prefixSum[myStartIdx + j] = mySum;
      }
      partitionSums[tid] = mySum;
      // Phase 1 done
      ...
```

Parallel Prefix Sum using Single Monitor (2)

```
      ... continuation of threads' run method
      // Phase 1 done
      synchronized(ParallelPrefixSumWait.class) { // thread tid enters monitor
        threadsFirstPhaseDone++;                   // and updates 'done' counter
        if (threadsFirstPhaseDone == NUM_THREADS) {
          ParallelPrefixSumWait.class.notifyAll();  // last thread wakes up everybody
        }
        while (!startThirdPhase) { // thread tid now waits until 3rd phase begins
          try {
            ParallelPrefixSumWait.class.wait();
          } catch(InterruptedException e) { e.printStackTrace(); }
        }
      } // thread tid leaves monitor

      // Phase 3: (PARALLEL): update prefixSum with partition offsets
      if (tid != 0) { // thread zero has nothing to do because its offset is zero
        for (int j = 0; j < elementsPerThread; j++) {  // iterate over thread's partition of the data
          prefixSum[myStartIdx + j] += partitionSums[tid - 1]; // update with partition offset
        }
      }
    });
    threads[i].start();
  }

  ...
```

Parallel Prefix Sum using Single Monitor (3)

```
      ... continuation of main thread

  // wait until threads are done with phase 1
  synchronized (ParallelPrefixSumWait.class) {
    try {
        ParallelPrefixSumWait.class.wait();
    } catch(InterruptedException e) { e.printStackTrace(); }

    // Phase 2 (SEQUENTIAL): main thread computes prefix sum on partitionSums
    for (int i = 1; i < NUM_THREADS; i++) {
      partitionSums[i] = partitionSums[i] + partitionSums[i - 1];
    }

    // Signal all threads the start of Phase 3
    startThirdPhase = true;
    ParallelPrefixSumWait.class.notifyAll();
  }

  // main-thread joins all threads
  for (Thread t : threads) {
    try {
      t.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Exercise 4: Problem 2

Consumer-Producer: Synchronization between Producing and Consuming Thread

Callables and Futures

Callable<V> Interface

- Callable can be considered a cousin of Runnable.
- They both can be used to describe a task.
- Difference:
  - A callable returns a value.
  - A callable may throw an exception.

Callable

```
@FunctionalInterface
public interface Callable<V> {
  /**
   * Computes a result, or throws an exception
   * if unable to do so.
   *
   * @return computed result
   * @throws Exception if unable to compute a result
   */
  V call() throws Exception;
}
```

Runnable

```
@FunctionalInterface
public interface Runnable {

  public void run();
}
```

Future<V> Interface

- `V get() throws InterruptedException, ExecutionException`
  Waits if necessary for the computation to complete, and then retrieves its result. Throws `ExecutionException` if an error occurred during the computation. Throws `InterruptedException` if current thread was interrupted while waiting.

- `V get(long timeout, TimeUnit unit)`
  `throws InterruptedException, ExecutionException, TimeoutException`
  Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available. Throws `ExecutionException` if an error occurred during the computation, `InterruptedException` if current thread was interrupted while waiting, and `TimeoutException` if operation timed out.

- `boolean isDone()`: Returns `true` if the task is complete (normally or with an exception).
- `boolean cancel(boolean mayInterruptIfRunning)`: Attempts to cancel task. If argument is `true` then the task should be interrupted, otherwise, it is allowed to continue.
- `boolean isCancelled()`: Returns `true` if this task was cancelled before it completed normally.

Submitting a `Callable` as Task to an `ExecutorService`

The executor service contains several methods take a Callable as argument or return Future.

- `<T> Future<T> submit(Runnable task, T result)`: Execute the specified `Runnable` (like in a `Thread`) and return the specified `result` in a `Future` that completes when the task is done.

- `Future<?> submit(Runnable task)`: Execute the specified `Runnable` (like in a `Thread`) and return a `Future` that completes with the value `null` when the task is done.

- `<T> Future<T> submit(Callable<T> task)`: Execute the specified `Callable` that produces a result of type `T` and return it as a `Future`.

Example: Submit Callable on ExecutorService returns Future

```java
public class CallableFuture {
  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
    Callable<Integer> callable = new Callable<Integer>() {
      @Override
      public Integer call() {
        try {
          Thread.sleep(5000); // sleep for 5 seconds
        } catch (InterruptedException e) { e.printStackTrace(); }
        System.out.println("[in callable] returning 42");
        return 42;
      }
    };
    Future<Integer> future = executor.submit(callable);
    System.out.println("[in main] callable submitted.");
    System.out.println("[in main] waiting for future to complete...");
    try {
      int result = future.get();
      System.out.println("[in main] got " + result + " from future.");
    } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }

    executor.shutdown();
    try { executor.awaitTermination(1, TimeUnit.MINUTES);}
    catch(InterruptedException ignored) { }
  }
}
```

Output

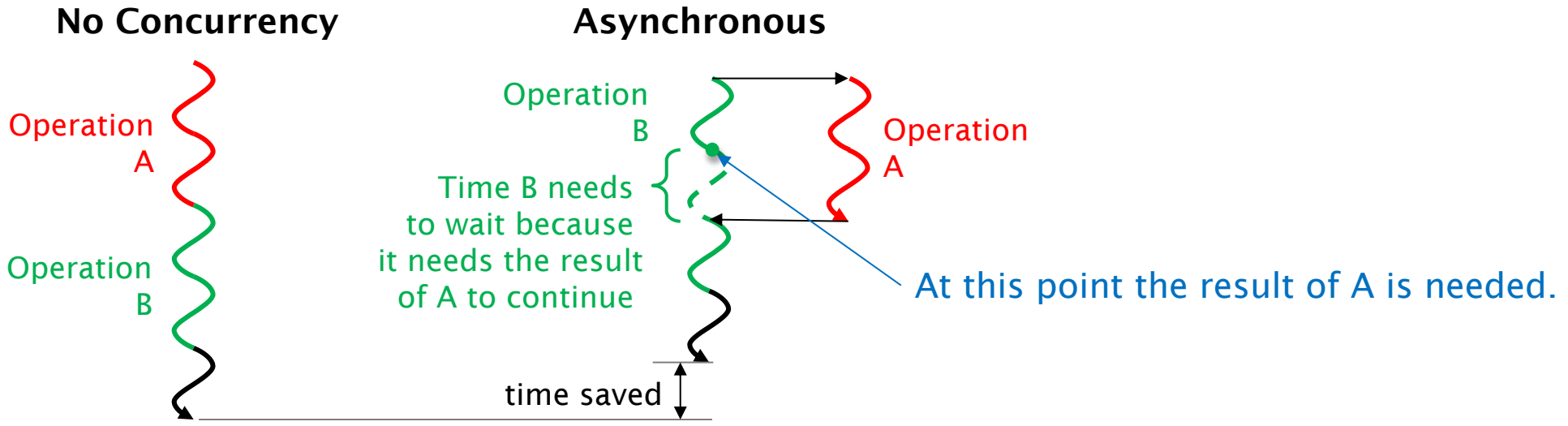
```
[in main] callable submitted.
[in main] waiting for future to complete...
[in callable] returning 42
[in main] got 42 from future.
```

Example: Submit `Callable` as Lambda

```
public class CallableFuture {
  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
    Callable<Integer> callable = () -> {
      try {
        Thread.sleep(5000); // sleep for 5 seconds
      } catch (InterruptedException e) { e.printStackTrace(); }
      System.out.println("[in callable] returning 42");
      return 42;
    };
    Future<Integer> future = executor.submit(callable);
    System.out.println("[in main] callable submitted.");
    System.out.println("[in main] waiting for future to complete...");
    try {
      int result = future.get();
      System.out.println("[in main] got " + result + " from future.");
    } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }

    executor.shutdown();
    try { executor.awaitTermination(1, TimeUnit.MINUTES);}
    catch(InterruptedException ignored) { }
  }
}
```

Asynchronous Operations using Callables/Futures

- Callables/Futures are a simple form of asynchronous programming.
- The operation in the `Callable` is executed as a separate tasks.
- The thread submitting the task does not need to wait for the task to complete, instead it can continue until it actually needs the task's result.
- Caller gets a `Future` from which the tasks result can be obtained later.
- There are two possibilities when caller actually needs the result from the task:
  - Task is complete, `Future` returns result immediately → Time is saved as nobody had to wait.
  - Task is not yet complete, caller needs to wait until task finishes and result becomes available.



Application Use-Cases of Callable/Future

- Essentially two reasons:
  - I/O-bound tasks: Issue I/O requests that take time to complete.
    - Concurrent processing of tasks.
    - Reading/writing data from disks, over the network.
    - File I/O, Web and REST APIs, Database Requests.
    - Allows hiding of I/O latency. The the I/O is performed, that main thread can go on and do other things. Waiting is only necessary if the I/O time cannot be filled other operations do be done by the main thread.

  - CPU-bound tasks: Issue compute tasks that are executed in parallel.
    - Parallel processing on multi-core systems.
    - Independent computations, i.e., tasks that do not need coordination.
    - "Divide and conquer" problems.
      (Java's Fork/Join Pools are a more general extension that permit a recursive tasks splitting of tasks)

Exercise 4: Problem 3

Callables and Futures:

Calling Web Services to get temperatures of various places in Switzerland concurrently and compute the average temperature.

Asynchronous Programming

- Synchronous Programming = Operations are performed synchronously in specified sequence in which the subsequent operation does not start before the the previous is complete.

- Asynchronous Programming: dealing events and operations asynchronously, i.e.., not in specified program order.
  - Event-based Programming: Event Handlers, Callbacks
  - Asynchronous Computations: Perform Computations as soon as possible if there are no Data Dependencies.

Callback in JavaFX: Event Handler

```
public class CallbackJavaFX extends Application {

  @Override
  public void start(Stage primaryStage) {
    System.out.println("in start: " + Thread.currentThread());
    Button button = new Button();
    button.setText("Click Me!");
    button.setOnAction(new EventHandler<ActionEvent>() {

      @Override
      public void handle(ActionEvent event) {
        System.out.println("click!, in handle: " + Thread.currentThread());
      }
    });

    StackPane root = new StackPane();
    root.getChildren().add(button);
    Scene scene = new Scene(root, 300, 75);
    primaryStage.setScene(scene);
    primaryStage.show();
  }

  public static void main(String[] args) {
    System.out.println("in main: " + Thread.currentThread());
    Application.launch(args);
  }
}
```

UI thread invokes callback to handle click event.

Output

thread name

priority

name thread group

```
in main: Thread[main,5,main]
in start: Thread[JavaFX Application Thread,5,main]
click!, in handle: Thread[JavaFX Application Thread,5,main]
```

→ All UI operations can be handled by one single thread while main thread sleeps. One thread handling UI events. **A single-CPU system is perfectly sufficient to run these programs.**

Callback are very popular in JavaScript (Node JS)

Asynchronous APIs: Caller needs to provide a callback function that will be invoked when the operation is complete (e.g., the data has been read)
Callback function will receive argument and error as argument.

Example: Reading a file in JavaScript

Callback function will be invoked if
file is read → content as string in argument `data`
or an error occurred → error in argument `err`.

```
fs.readFile(filename, 'utf8', function (err, data) {
  if (err) {
    console.log('Error reading file: ' + err);
  } else {
    console.log('File content: ', data);
  }
});
```

Asynchronous I/O is the key feature in Node JS. Server-side application that perform a many I/O operations can be executed without multi-threading but with asynchronous I/O provided by the operating system and with callbacks.

"Callback Hell" is also very popular in JavaScript

- Callback Hell = Nesting of callbacks yields unreadable code (in particular with error handling)
  Can be addressed by using Promises and Generators in ES6 (by using Q or BlueBird libraries).
- **Example**: Open all files in a directory, read the single text line they contain, parse it as an integer, and sum up the value. Finally, print the value to the console.
-

```
var count = 0;
var sum = 0;
function callback() { console.log('sum = ', sum); }
fs.readdir(dir, function (err, files) {   // 1st callback
  if (err) { // 1st error handler
    console.log('Error finding files: ' + err);
  } else {
    files.forEach(function (filename, fileIndex) { // 2nd callback
      console.log(filename);
      fs.readFile(dir + path.sep + filename, 'utf8', function(err, data) { // 3rd callback
        if (err) {// 2nd error handler
          console.log('Error reading file: ' + err);
        } else {
          sum += parseInt(data);
          count += 1;
          if (count == files.length) callback(); // 4th "callback"
        }
      });
    });
  }
});
```

?

We want to execute operations asynchronous but

- … we also want to chain operations.

- We do not want to deal with threads and synchronization.
- We do not want to specify *when* operations are performed, we only want to have to specify the dependencies (if at all).

Example:

- Do Operation A
- Do Operation B
- Do Operation C after A and B is complete
- Do Operation D after C is complete

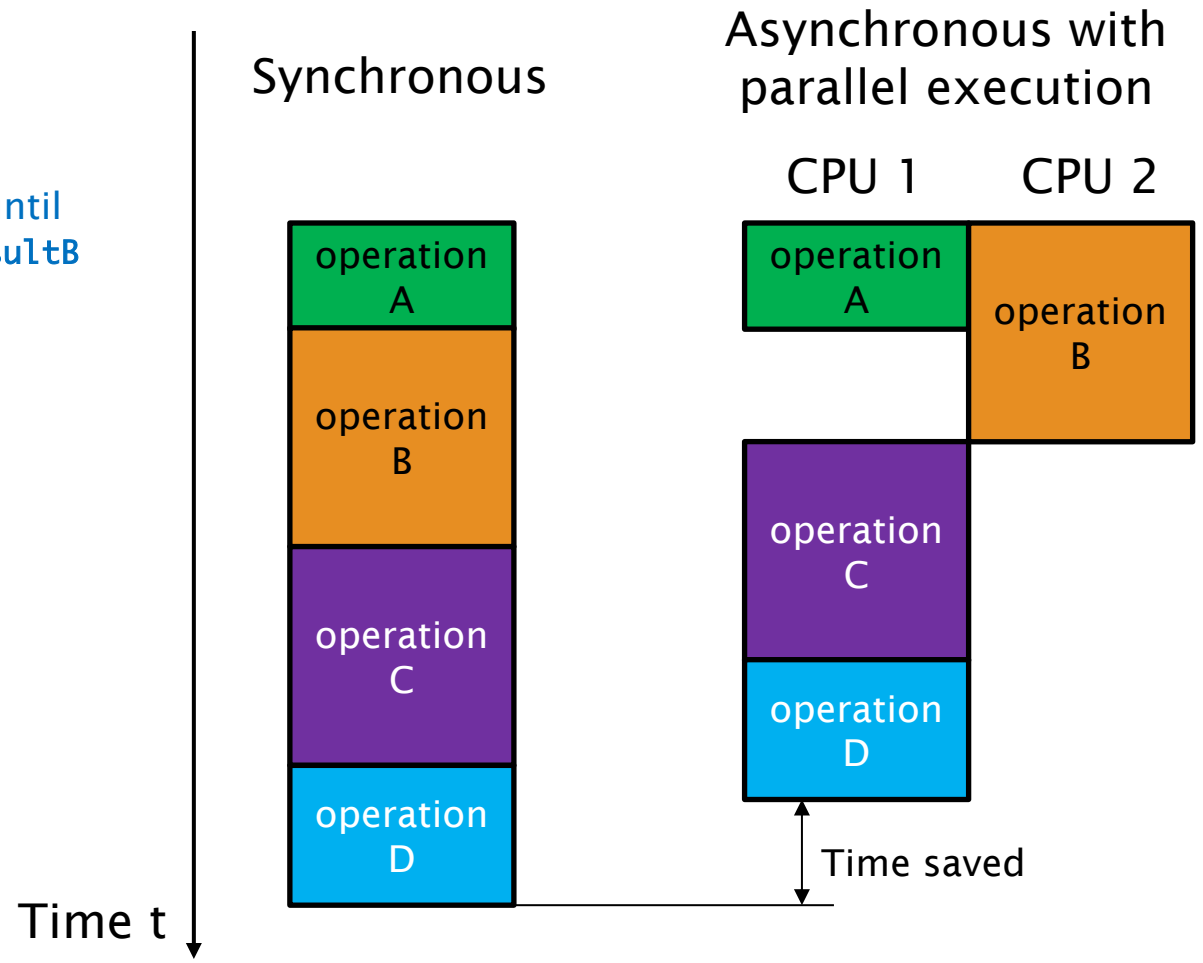
Executing Operations Synchronously and Asynchronously

Example:

```
resultA = operationA()
resultB = operationB()

resultC = operationC(resultA, resultB)

resultD = operationD(resultC)
```

Data Dependencies:
`operationC` cannot start until values of `resultA` and `resultB` are available.

Data Dependency:
`operationD` cannot start until value of `resultC` is available.



Asynchronous Execution in Super-Scalar CPUs

- Asynchronous execution happens in modern CPUs already!
- Intel Skylake/Kaby Lake can issue up to 8 Instructions/cycle (up to 8 μOPs/cycle)
  - Skylake/Kaby Lake cores have 4 ALUs that could execute Integer operations

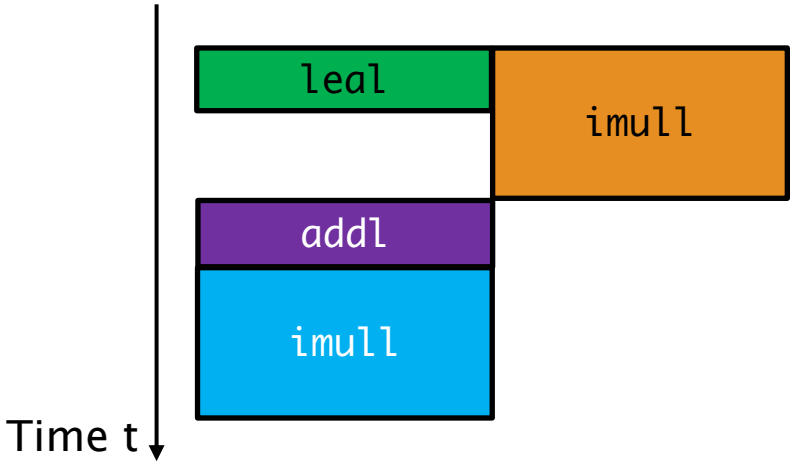
```
resultA = operationA()
resultB = operationB()

resultC = operationC(resultA, resultB)

resultD = operationD(resultC)
```

Example: x86 Assembler code for a similar sequence of operations

```
leal   (%rsi,%rdi), %eax   # eax <- rsi + rdi
imull  %esi, %edi          # edi <- esi * edi
addl   %eax, %edi          # edi <- eax + edi
imull  $42, %edi, %eax     # eax <- 42 * edi
```

data dependencies

data dependencies



Limitations of Future<T>

- Futures cannot be explicitly be completed.
- Futures cannot handle exceptions themselves.
- You cannot get notified when a Future completes.
- Futures cannot be chained.

`CompleteableFuture<T>` from Java 8 to the rescue.

Asynchronous Programming using CompletableFutures<T> since Java 8

```
CompletableFuture<Integer> opA = CompletableFuture.supplyAsync(() -> { // operation A
  System.out.println("OpA started.");
  try { Thread.sleep(10000); } catch(InterruptedException e) { e.printStackTrace(); }
  System.out.println("OpA done.");
  return 11; // operation A returns 11
});
CompletableFuture<Integer> opB = CompletableFuture.supplyAsync(() -> { // operation B
  System.out.println("OpB started.");
  try { Thread.sleep(5000); } catch(InterruptedException e) { e.printStackTrace(); }
  System.out.println("OpB done.");
  return 7;  // operation B returns 7
});
CompletableFuture<Integer> opC = opA.thenCombineAsync(opB, (a, b) -> { // C runs after A and be are complete
  System.out.println("OpC started.");
  try { Thread.sleep(2000); } catch(InterruptedException e) { e.printStackTrace(); }
  System.out.println("OpC done.");
  return a + b;
});
CompletableFuture<Integer> opD = opC.thenApplyAsync((x) -> { // D runs after C is complete
  System.out.println("OpD started.");
  try { Thread.sleep(3000); } catch(InterruptedException e) { e.printStackTrace(); }
  System.out.println("OpD done.");
  return 42 * x;
});
try {
  int resultD = opD.get();
  System.out.println("result OpD: " + resultD);
} catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
```

Takes results of operations A and B and runs Lambda as another asynchronous task.

Takes results of C and runs Lambda as another asynchronous task.

Main thread waits until result of operation D is available.

Synchronized Collections

Synchronized Collections

- Java collection such as Lists, Sets, or Maps are **not** thread-safe.
  - Race condition may occur when the methods of the collection classes are called by multiple threads.
- **Rationale**: Mutual exclusion (Monitors) is too costly to be added just in case someone wants to use a collection with multiple threads.
- If you need thread-safe collections, wrap them by thread-safe wrapper classes. This approach is called Decorator Pattern (later in BTI7081 Software Engineering and Design).

- The class `java.util.Collections` provides several static methods to construct thread-safe wrappers:
  - `Collection<T> synchronizedCollection(Collection<T> c)`
  - `List<T> synchronizedList(List<T> list)`
  - `Set<T> synchronizedSet(Set<T> s)`
  - `SortedSet<T> synchronizedSortedSet(SortedSet<T> s)`
  - `Map<K,V> synchronizedMap(Map<K,V> m)`
  - `SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)`
  - `NavigableMap<K,V> synchronizedNavigableMap(NavigableMap<K,V> m)`

Example: Accessing non-thread safe `Collection` using multiple Threads (BAD)

```java
public class UnsynchronizedCollections {
  private static final int NUM_THREADS = 8;
  public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();

    Thread[] threads = new Thread[NUM_THREADS];
    long startTime = System.nanoTime();
    for (int i = 0; i < NUM_THREADS; i++) { // fork worker threads
      threads[i] = new Thread(() -> {
        for (int j = 0; j < 1000000; j++) {
          list.add(j);
        }
      });
      threads[i].start();
    }
    for (Thread t : threads) { // join worker threads
      try { t.join(); } catch (InterruptedException ignored) { }
    }
    long stopTime = System.nanoTime();
    System.out.println("length: " + list.size());
    System.out.format("duration: %4.3f sec\n", (stopTime - startTime)*1e-9);
  }
}
```

Not a thread-safe implementation!

Output

```
length: 1816611
duration: 0.688 sec
```

Result is incorrect!
→ Race Condition.

Example: Wrapping Collection with thread-safe Wrapper

```java
public class SynchronizedCollections {
  private static final int NUM_THREADS = 8;
  public static void main(String[] args) {
    List<Integer> list = Collections.synchronizedList(new LinkedList<>());

    Thread[] threads = new Thread[NUM_THREADS];
    long startTime = System.nanoTime();
    for (int i = 0; i < NUM_THREADS; i++) { // fork worker threads
      threads[i] = new Thread(() -> {
        for (int j = 0; j < 1000000; j++) {
          list.add(j);
        }
      });
      threads[i].start();
    }
    for (Thread t : threads) { // join worker threads
      try { t.join(); } catch (InterruptedException ignored) { }
    }
    long stopTime = System.nanoTime();
    System.out.println("length: " + list.size());
    System.out.format("duration: %4.3f sec\n", (stopTime - startTime)*1e-9);
  }
}
```

Output

```
length: 8000000
duration: 5.245 sec
```

Result is correct, but execution took 10x as long!
→ Mutual Exclusion is expensive

Summary

- Threads are used to represent concurrent flows of execution in programs.
- Threads can be run in parallel if there is more than one processor available.

- Worker threads are forked-off from the main thread.
  - Specify thread method via explicit implementation of `Runnable` or via Lambda.
- Threads may need to be coordinated:
  - Main thread joining worker threads
  - Order access to shared resources and variables
  - Wait for other threads to reach a certain state (`wait/notify/notifyAll`)

- The creation and destructions of threads is expensive
  → recycle threads, i.e., put them in a pool after they have been used → Thread Pools (`ExecutorService`)

- Java collections class are not thread-safe, use special wrapper class from Collections when you need thread-safe collections.

Synchronization

- Data Races: situation if the relative ordering of threads that access shared variables affect the program's correctness.
- Mutual Exclusion: only one thread is allowed to be inside a critical section of the code at any point in time.
- Monitors: Primitive that provides mutual exclusion using `synchronized` methods and `synchronized` code blocks.
- Locks: Fine-graind mutual exclusion using with a `lock()` and `unlock()` primitive. Do not forget to unlock in every possible code path.

Asynchronous Execution

- Callables: Implementation of `V call()`.
- Can be asynchronous execution of code, result is provided as a `Future<V>`.
- `Future<V>` yields value if asynchronous execution is complete, otherwise, blocks.

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Object-Oriented Programming 2

Java Streams

Prof. Dr. Annett Laube
Prof. Dr. René Müller

FS 2018

Book Chapters:

- Chapters 4—7: "Functional-style data processing" in Raoul-Gabriel Urma et al., "Java 8 in Action: Lambdas, streams, and functional style programming", Manning Publications, 2014.

Outline Topic 10: JavaStreams

Feature of Java 8

- Functional Interfaces, Method References, and Optional<T> Monad
- Manipulate Flows of Data in a Declarative Way
- Filter, Map, FlatMap, Reductions
- Parallel Streams

Watch out!

Java 8 **Streams** not be confused with Java **I/O Streams** (from the java.io package).

They are different things!

Motivation

Streams as Internal Iterators

Example: Processing a CSV File

- CSV File: Comma-separated Values (text) File
- Given this CSV file containing persons:

User ID
Name
Age
Gender: Female (F), Male (M)

```
12121,Brian,47,M
12223,Adam,32,M
12231,Claire,30,F
12326,Eve,54,F
12532,Frances,44,F
13421,Carl,62,M
12321,Bob,32,M
12512,Alice,44,F
12345,Charlie,29,M
```

persons.csv

- **Task**: Determine the average age the females persons.

Solution 1: Process CSV File using an Iterator of Text Lines

```java
public class ProcessTextFileIterator {

  private static final int USER_ID_COLUMN = 0;
  private static final int NAME_COLUMN = 1;
  private static final int AGE_COLUMN = 2;
  private static final int GENDER_COLUMN = 3;

  public static void main(String[] args) throws FileNotFoundException {

    TextFileLineIterator lines = new TextFileLineIterator("persons.csv"); // see next slide
    int count = 0;
    double ageSum = 0;
    while (lines.hasNext()) {      // iterate over every line in the file
      String line = lines.next();  // get next text line
      String[] columns = line.split(","); // split text line at the comma
      if (columns[GENDER_COLUMN].toUpperCase().equals("F")) {  // filter out females only
        ageSum += Double.parseDouble(columns[AGE_COLUMN]);     // extract and parse 'age' column
        count++;
      }
    }
    System.out.println("females:                 " + count);
    System.out.println("average age of females: " + (ageSum/count));
  }
}
```


Download ProcessTextFileIterator.java, TextLineFileIterator.java, and persons.csv from Moodle.

Solution 1: TextFileLineIterator Class (1)

```java
/** An iterator that open a text file and returns the file line-by-line. */
public class TextFileLineIterator implements Iterator<String> {
  private BufferedReader reader;
  private String line = null;

  public TextFileLineIterator(String fileName) throws FileNotFoundException {
    reader = new BufferedReader(new FileReader(fileName));
  }

  @Override
  public boolean hasNext() {
    if (line == null) {
      tryFetchingNextLine();
    }
    return line != null;
  }

  ... // see next slide
```

Solution 1: TextFileLineIterator Class (2)

```
  ... // see previous slide

  @Override
  public String next() {
    if (line == null) {
      tryFetchingNextLine();
    }
    if (line == null) {
      throw new NoSuchElementException();
    }
    String result = line;
    line = null;
    return result;
  }

  private void tryFetchingNextLine() {
    try {
      line = reader.readLine();
    } catch(IOException e) { e.printStackTrace(); }
    if (line == null) {
      try { reader.close(); } catch (IOException ignored) { }
    }
  }
}
```

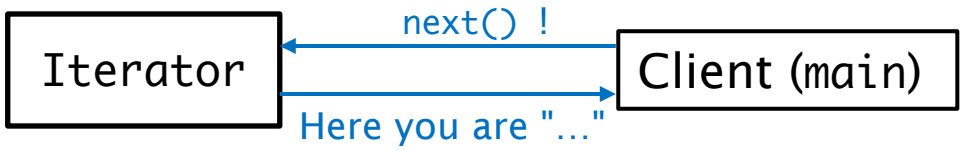
Analysis of Solution 1

Output

```
females:                    4
average age of females: 43.0
```

```
TextFileLineIterator lines = new TextFileLineIterator("persons.csv");
int count = 0;
double ageSum = 0;
while (lines.hasNext()) {      // iterate over every line in the file
  String line = lines.next();  // get next text line
  String[] columns = line.split(","); // split text line at the comma
  if (columns[GENDER_COLUMN].toUpperCase().equals("F")) {  // filter out females only
    ageSum += Double.parseDouble(columns[AGE_COLUMN]);     // extract and parse 'age' column
    count++;
  }
}
```

- Client fetches line by line from the iterator.
  - Client is in control. When requested, iterator needs to provide the next element.
  - This also called an External Iterator because the Iterator is not in control.



- Operations: splitting string, filtering female records, parsing and summation of age column are implemented in the client as *"regular" Java code*, in what we call, imperative style.

Code Improvement: Client with a for-each loop?

We could try to improve the readability of the program by using a for-each loop.

```
TextFileLineIterator lines = new TextFileLineIterator("persons.csv");
int count = 0;
double ageSum = 0;
for (String line : lines) {      // iterate over every line in the file
  String[] columns = line.split(","); // split text line at the comma
  if (columns[GENDER_COLUMN].toUpperCase().equals("F")) {  // filter out females only
    ageSum += Double.parseDouble(columns[AGE_COLUMN]);     // extract and parse 'age' column
    count++;
  }
}
```

For-each loop

Why doesn't this work?

```
Error:(17, 24) java: for-each not applicable to expression type
  required: array or java.lang.Iterable
  found:    ch.bfh.ti.oop2.streams.TextFileLineIterator
```

→ We need to provide an **Iterable** and not an **Iterator**!

Exercise 5: Problem 1

Create a `TextFileLineIterable` such that you can use it in the for-each foop.

Solution 2: Java 8 Streams and forEach

Java 8 Streams offers another possibility to implement the client:

```
public class ProcessTextFileStreamsForEach {

  private static final int USER_ID_COLUMN = 0;
  private static final int NAME_COLUMN = 1;
  private static final int AGE_COLUMN = 2;
  private static final int GENDER_COLUMN = 3;

  private static int count = 0;
  private static double ageSum = 0;

  public static void main(String[] args) throws IOException {

    Stream<String> stream = Files.lines(Paths.get("persons.csv"));
    stream.forEach(line -> {  // internal iterator: stream invokes this lambda expression for every text line
      String[] columns = line.split(","); // split text line at the comma
      if (columns[GENDER_COLUMN].toUpperCase().equals("F")) {  // filter out females only
        ageSum += Double.parseDouble(columns[AGE_COLUMN]);     // extract and parse 'age' column
        count++;
      }
    });
    System.out.println("females:                 " + count);
    System.out.println("average age of females: " + (ageSum/count));
  }
}
```

Produces a Stream of String items, one item per text line in the file.

forEach(line -> {…}) instructs the stream to invoke the specified lambda function on every element.

Analysis: Solution 2

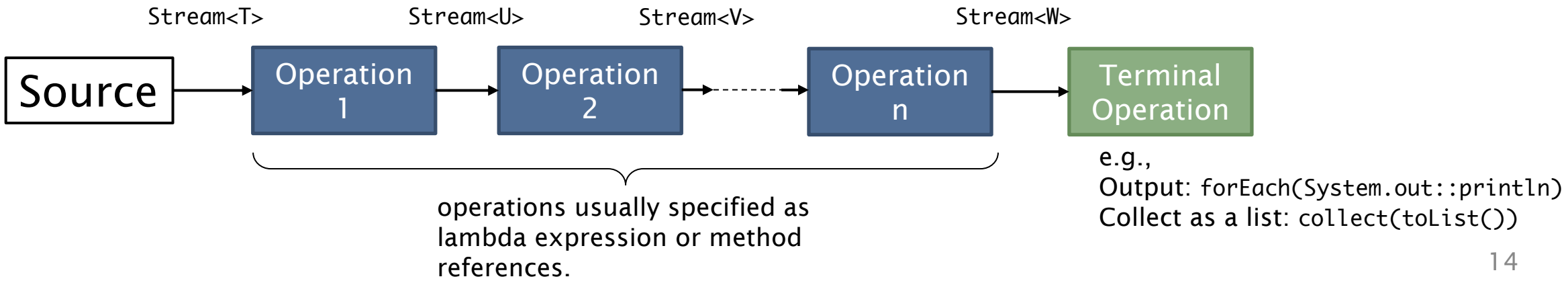
```
Stream<String> stream = Files.lines(Paths.get("persons.csv"));
stream.forEach(line -> {  // internal iterator: stream invokes this lambda expression for every text line
  String[] columns = line.split(","); // split text line at the comma
  if (columns[GENDER_COLUMN].toUpperCase().equals("F")) {  // filter out females only
    ageSum += Double.parseDouble(columns[AGE_COLUMN]);     // extract and parse 'age' column
    count++;
  }
});
```

- In this solution, the stream is in charge.
- *It* invokes the lambda expression, specified in `stream.forEach(..)`, on every stream item.
  - Client registers callback function.
  - Inversion of Control, aka Hollywood Principle: "Don't call us, we'll call you".
- The Stream is an Internal Iterator.
  - Internal = it keeps the control, i.e., determines when to process an element.
  - External = client has control (see Solution 1)

- The processing is still performed in regular Java code, now just inside a lambda expression. Code is *still imperative*, worse, we now also need to modify global variables!

Java 8 Streams

- Java 8 introduced the concept of a Stream.
- A Stream represents a sequence of elements (similar to an `Iterable`).
- Each stream originates from a source.
- The main purpose of a stream is to iterate through elements and to perform operations on the elements.
- The Stream API allows a declarative (functional) description and composition of the operations applied on the stream. *as opposed to imperative*
- A Streams are Monads: Monads are used in functional programming express sequences of steps.
- The chained operations on a Stream form a Pipeline:



Solution 3: Java 8 Streams as Functional Pipeline

```
public class ProcessTextFileStreamsFunctional {

  private static final int USER_ID_COLUMN = 0;
  private static final int NAME_COLUMN = 1;
  private static final int AGE_COLUMN = 2;
  private static final int GENDER_COLUMN = 3;

  public static void main(String[] args) throws IOException {

    Stream<String> stream = Files.lines(Paths.get("persons.csv"));

    DoubleSummaryStatistics stats = stream
        .map(line -> line.split(",")) // split text line at the comma
        .filter(columns -> columns[GENDER_COLUMN].toUpperCase().equals("F")) // filter out females
        .mapToDouble(columns -> Double.parseDouble(columns[AGE_COLUMN])) // parse age
        .summaryStatistics(); // terminal operator computes statistics min/max/count/sum

    System.out.println("females:                 " + stats.getCount());
    System.out.println("average age of females: " + stats.getAverage());
  }
}
```

Creates a `Stream<String>` from the file. The stream contains one `String` element for every text line in the file.

Composed Operations on Stream

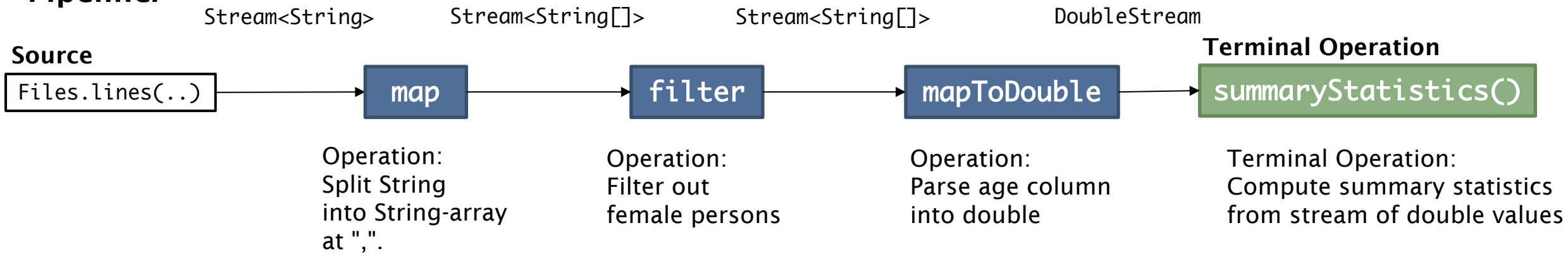
Note the formatting style here for method chaining, i.e., chaining the operation methods.

Analysis: Solution 3

"Look Ma, no 'for' loops!"

```
DoubleSummaryStatistics stats = Files.lines(Paths.get("persons.csv"))
  .map(line -> line.split(",")) // split text line at the comma
  .filter(columns -> columns[GENDER_COLUMN].toUpperCase().equals("F")) // filter out females
  .mapToDouble(columns -> Double.parseDouble(columns[AGE_COLUMN])) // parse age
  .summaryStatistics(); // terminal operator computes statistics min/max/count/sum
```

Pipeline:



- The operations are specified as lambda expressions.
- The types of the streams change throughout the Pipeline. There is a specialized class for Streams-of-Double that provides additional operations, e.g., statistics.
- There is no "`if`" anymore. → This is a declarative, functional definition of the processing steps. Declarative: we specify *what to do* not but not *how*.
- Processing is lazy. Nothing happens until there is a terminal operation that actually reads the stream.

Streams from Collections

- Streams can also be created from Collections.

Example:

```
public class StreamsFromCollections {
  public static void main(String[] args) {
    // Create a list of random numbers.
    Random rnd = new Random();
    List<Integer> numbers = new ArrayList<>();
    for (int i = 0; i < 1_000_000; i++) {
      numbers.add(rnd.nextInt(1001));
    }

    int sum = numbers.stream()     // creates Stream<Integer> from List<Integer>
        .filter(n -> n % 2 == 0)   // filters even numbers
        .mapToInt(n -> n - 500)    // subtracts 500 from each number, returns IntStream
        .sum();                    // terminal operation: sums up all elements
    System.out.println(sum);
  }
}
```

Create a Stream from a Collection.

Streams can also be Parallel

"*Look Ma, No Threads*!"

Code written in a functional or declarative programming style, in general, is much easier to parallelize than imperative code. Very often, it can even be parallelized automagically.
Magically = change `stream()` to `parallelStream()`

Example:

```
public class StreamsFromCollections {
  public static void main(String[] args) {
    // Create a list of random numbers.
    Random rnd = new Random();
    List<Integer> numbers = new ArrayList<>();
    for (int i = 0; i < 1000000; i++) {
      numbers.add(rnd.nextInt(1001));
    }

    int sum = numbers.parallelStream()    // creates Stream<Integer> from List<Integer>
        .filter(n -> n % 2 == 0)  // filters even numbers
        .mapToInt(n -> n - 500)   // subtracts 500 from each number, returns IntStream
        .sum();                   // terminal operation: sums up all elements
    System.out.println(sum);
  }
}
```

Create a Stream from collection that is processed in parallel.

First, we need to cover some new features in Java 8…

… that are needed take full advantage of Streams:

- Functional Interfaces
- Method References
- `Optional<T>` Monad

Functional Interfaces in Java 8

Lambda Expressions as Interfaces

Problem: A Method that Accepts a Function as Argument

- We want to provide a function to a method.
- The method requires a very specific function with a specific number of arguments and types.

Motivating Example: We want a `TextFileReader` that reads and processes the entire text file.
It takes a function and applies it on every text line in the file and returns a **list of the results**.

```
TextFileReader reader = new TextFileReader("persons.csv");

// Read file as a list of text lines: identity function
List<String> stringList = reader.getListAs(line -> line);

// Read file as a list of integers: parses age column of the CSV file
List<Integer> ageList = reader.getListAs(line -> Integer.parseInt(line.split(",")[2]));

// Read file as a list of Booleans: true if the record corresponds to a female person
List<Boolean> isFemaleList = reader.getListAs(
  line -> line.split(",")[3].toUpperCase().equals("F"));
```

Age is 3nd column

Gender is 4th column

Signature of `getAsList` Method?

- What is the signature of the `getAsList` Method?
- We need a way to express that the caller must provide a function that
  - has one single argument that is of type String
  - but can return a value of any type R.

- Unfortunately, the types of (lambda) functions cannot be directly expressed in Java.
- Java is not a functional programming language that truly supports higher-order functions (in contrast what many claim!)

- In Scala, we could simply express the method signature as follows:

```
def getListAs[R](function: String => R): List[R]
```

  - The method is a generic method in type R.
  - As argument, the method takes a function that maps from a `String` to an R.
  - It returns a List of type R.

- Java 8 introduced a "hack" that can be used to express the type of a function argument using a special Interface, called a Functional Interface.

Java's Solution

```
// Define a generic interface for the function which takes an argument of type T and returns a value of type R
interface UnaryFunction<T, R> {
  R apply(T t);
}

public class TextFileReader {
  private final String fileName;
  public TextFileReader(String fileName) {
    this.fileName = fileName;
  }

  // getListAs has an argument of type of the interface.
 public <R> List<R> getListAs(UnaryFunction<? super String, ? extends R> function)
                                                    throws FileNotFoundException {
    List<R> list = new ArrayList<R>();
    TextFileLineIterator it = new TextFileLineIterator(fileName);  // reuse class from slide (7)
    while (it.hasNext()) {
      String line = it.next();        // get next text line
      list.add(function.apply(line)); // invoke the function on every element
    }
    return list;
  }
}
```

Note the Type Bounds!
Supplied class that implements the **apply** method
may accept something **more general** than String.
may provide something **more specific** than R.

(Remember the FruitBasket from Topic 7)

Invoking getListAs before Java 8

- Implement the interface as an anonymous inner class:

```
TextFileReader reader = new TextFileReader("persons.csv");
```

f: String → String

```
List<String> stringList = reader.getListAs(new UnaryFunction<String, String>() {
  @Override
  public String apply(String s) {
    return s;
  }
});
```

f: String → Integer

```
List<Integer> ageList = reader.getListAs(new UnaryFunction<String, Integer>() {
  @Override
  public Integer apply(String s) {
    return Integer.parseInt(s.split(",")[2]);
  }
});
```

f: String → Boolean

```
List<Boolean> isFemaleList = reader.getListAs(new UnaryFunction<String, Boolean>() {
  @Override
  public Boolean apply(String s) {
    return s.split(",")[3].toUpperCase().equals("F");
  }
});
```

Functional Interfaces (since Java 8)

- A functional interface is an interface that specifies exactly one abstract method.
- Use the annotation `@FunctionalInterface` **before** the declaration of the interface.
  - This will cause a compiler error if the interface does not satisfy this restriction, i.e., the compiler reports an error if the interface contains more than one abstract method or none.

```
@FunctionalInterface
interface UnaryFunction<T, R> {
  R apply(T t);
}
```

- **The Trick:** A matching lambda expression is a valid implementation of a functional interface.

```
// Instead of ...
List<Integer> ageList = reader.getListAs(new UnaryFunction<String, Integer>() {
  @Override
  public Integer apply(String s) {
    return Integer.parseInt(s.split(",")[2]);
  }
});
// ... we can write more concisely:
List<Integer> ageList = reader.getListAs(line -> Integer.parseInt(line.split(",")[2]));
```

This lambda expression takes a **String** and returns an **int** and, thus, matches the **apply** method in the **UnaryFunction** Interface (considering Java's Autoboxing).

- The functional interface crutch allows us to define a method that accepts lambda expressions with a specific number of arguments, specific argument and return types.

Invoking getListAs since Java 8

```
TextFileReader reader = new TextFileReader("persons.csv");

List<String> stringList = reader.getListAs(line -> line); // identity function

List<Integer> ageList = reader.getListAs(
  line -> Integer.parseInt(line.split(",")[2]));

List<Boolean> isFemaleList = reader.getListAs(
  line -> line.split(",")[3].toUpperCase().equals("F"));


List<String> columnsList = reader.getListAs(line -> line.split(","));
// Error:(36, 48) java: incompatible types:
//     inference variable R has incompatible bounds
//     equality constraints: java.lang.String
//     lower bounds: java.lang.String[]
```

The method returns a String[] but getListAs() expects a String result.

Predefined Functional Interfaces in `java.util.function`

- The Java standard library provides a number pre-defined Functional Interfaces.
- These are usually sufficient such that you don't have to define your own functional interface. Instead of defining our own interface `UnaryFunction`, we could have simply used `Function<T,R>`.

Functional Interface	Function Descriptor	Primitive specializations
`Predicate<T>`	`T => boolean`	`IntPredicate, LongPredicate, DoublePredicate`
`Consumer<T>`	`T => void`	`IntConsumer, LongConsumer, DoubleConsumer`
`Function<T,R>`	`T => R`	`IntFunction<R>, IntoToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>`
`Supplier<T>`	`() => T`	`BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier`
`UnaryOperator<T>`	`T => T`	`IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator`
`BinaryOperator<T>`	`(T, T) => T`	`IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator`
`BiPredicate<L,R>`	`(L, R) => boolean`	
`BiConsumer<T,U>`	`(T, U) => void`	`ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>`
`BiFunction<T,U,R>`	`(T, U) => R`	`ToIntBiFunction<T,U>, ToLongBiFunction<T,U>, ToDoubleBiFunction<T,U>`

Method References Java 8

Passing references to methods as arguments

Method References in Java 8

- Consider this example: We want to convert the text line to an upper-case String.

```
TextFileReader reader = new TextFileReader("persons.csv");
List<String> upperCaseLine = reader.getListAs(line -> line.toUpperCase());
System.out.println(upperCaseLine);
```

- We define a lambda that takes a string (`line`) and invokes the method **`toUpperCase()`** on it.

- This seems redundant. Why do we have to specify a lambda to only invoke a method?
- Wouldn't it be easier to just say "invoke the method `toUpperCase()` on every element", i.e., `String`?

- Yes, and we can do so since Java 8! → Pass a method reference instead of a lambda expression.

```
List<String> upperCaseLine = reader.getListAs(String::toUpperCase);
System.out.println(upperCaseLine);
```

Method Reference with "`::`". Invokes the specified method on every element, just like the lambda would.

References to Static Methods

- Method references also works for static methods.

- **Example:** Assume we want to extract the name column (#2) and convert the Strings to upper-case and we *already* have a static method that does this:

```
private static String extractNameInUpperCase(String line) {
  return line.split(",")[1].toUpperCase();
}
```

- The signature of `extractNameInUpperCase` is equivalent to a lambda expression that takes a String and maps it into another String.

```
List<String> upperCaseName = reader.getListAs(line -> MyClass.extractNameInUpperCase(line));
```

- We can pass a reference to the static method it in lieu of the lambda expression.

```
List<String> upperCaseName = reader.getListAs(MyClass::extractNameInUpperCase);
```

The `Optional<T>` Monad

Writing robust software

Motivating Example

- What is correct type and behavior of `get(K key)` in a `Map<K, V>`?

- The method returns the value of type `V` for the key stored in the map.
- The problem is *what to do if the key does not exist*?

- **Possibilities:**
  - Return `null`
  - Throw a checked exception
  - Throw a unchecked exception
  - Do not call `get(K key)` but call `getOrDefault(K key, V defaultValue)` instead.
  - Return a dummy value of Type `V` (Null Pattern).
  - Return an `Optional<V>`.

Returning null

- **Seriously?**
  But this is exactly what `java.util.Map` requires implementations of this interface to do!

- Problems with this approach:
  - **`NullPointerException`s**!!!
  - Unless explicit tests are performed, each and every time after **`get(..)`** is called!
    - And then do what exactly? What is the action?
    - What if the map lookup occurs in a deeply nested method and this methods does not have sufficient context information to handle missing values on its own?
      - Method returns `null`?
      - `null` propagates up the calls stack until a method has the context to handle the missing value.
      - What if null check is omitted somewhere in-between?

- Any reasonable modern programming language does not even support `null`!

Throwing a Checked Exception

- The method then gets the signature:

```
public V get(K key) throws ElementNotFoundException
```

- Advantage: There is no `NullPointerException`.

- Question: Who catches the exception?
  - The method is caught in the method that knows how to handle missing entries.

- If the exception is thrown in the deeply nested method
  every method in the call stack up to the handling method needs to declare

```
throws ElementNotFoundException
```

  as well!

Throwing an **Unchecked** Exception

- The method keeps its original signature → no code changes ☺
  but can now throw an unchecked Exception (`Error` or `RuntimeException`)

- Who makes sure that the exception is handled at all?
  - Who even remembers that such an exception can be thrown from deep within the call stack?
  - An unhandled exception kills the thread.

- Besides, is a missing element cause for an exception?
- Exception: Shorthand for the phrase "exceptional event.", an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
  - Is a missing entry an exceptional event? (consider a Map used to cache objects)
    - Answer: Definitively not!
  - Does a missing entry require the same treatment as a ArrayOutOfBoundsException?
    - Answer: It depends.

get with Default Value

- Java 8 finally introduced a way to handle the situation of a missing element that is not totally broken:

```
V getOrDefault(Object key, V defaultValue)
```

- The `getOrDefault` method takes an additional argument for which the caller may supply a default value. This default value is returned if the key does not exist in the map.

- Problems:
  - Each and every time `getOrDefault` is called, the caller needs to provide a default value.
  - Possibly, this default object needs to be instantiated each and every time just for the rare case that a key does not exist. Instantiating objects of certain types `V` may be very expensive.
  - What if the method does not have sufficient context information to provide a suitable default object (same problem as before). Default value would need to be passed down the call stack.

Forcing Types V to Have a "Dummy" Value

- There are no abstract static methods in Java that could be used enforce the existence of a factory method in type V.
  By convention, one could demand the existence of an accessible default constructor but, remember, one cannot instantiate a generic type due to Type Erasure.

- Provide a factory that creates dummy (or default) values for Type V.
  ```
  V get(Object key, Supplier<V> dummyValueFactor)
  ```

- Advantage: You default/dummy object is only instantiated if the value is missing.

- Disadvantage: If deeply nested method may not know how to provide a suitable `Provider<V>`, thus, such a provider would need to be passed all the way down the call stack.

Enter the Optional<T> Monad

- Optional<T> is a container.
- An Optional<T> object may either
  - contain a non-null value of Type T
  - be empty, i.e., not contain any value.

- Class provides methods:
  - boolean isPresent(): Returns true if optional contains a value.
  - T orElse(T other): Returns value of optional if present, otherwise, returns other.
  - orElseGet(Supplier<? extends T> other): Returns value of optional if present, otherwise, uses specified supplier do produce a default value to return.
  - <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X: Returns value of optional if present, otherwise, use specified supplier to produce an exception to throw.
  - T get(): Returns value if present, otherwise, throws unchecked NoSuchElementException.
  - void ifPresent(Consumer<? super T> consumer): Applies consumer action if and only if optional contains a value.
- Non-empty instances are created using the static method Optional<T> of(T value).
- Empty instances are created using the static method Optional<T> empty().

get returning an Optional<V>

- The signature of the **get** Method should be:

```
public Optional<V> get(K key)
```

Analysis:

- A missing value is an object and not `null`! There is no `NullPointerException`.
- The result is an `Optional<V>` object and can be passed up the call stack.
- At the point in the code where the actual value is needed,
  - the programmer must perform a conscious act to extract the value. She cannot directly use the optional as value. The value needs to be extracted, thus, she must handle the scenario of an empty optional.
  - there is usually also sufficient context information to handle missing values.

- Optional also supports the functional `map` and `flatMap` operations:
  - `Optional<U> flatMap(Function<? super T, Optional<U>> mapper)`
  - `Optional<U> map(Function<? super T, ? extends U> mapper)`

Now,

… back to Streams!

`Stream` Interface and Operations

Properties of Streams

- Constant Memory: A stream does not store any element. It is not a data structure.
- Streams are immutable: An operation creates a new stream as a result, it does not modify the source stream.
- Lazy: Many stream operations can be performed lazily. The operations is not performed until the data is needed by a non-lazy downstream operator or a terminal operator. This opens up opportunities for optimization.
- No Bounds: Streams can be unbounded (infinite size). In contrast, Collections require memory and, thus, must have a finite size.
- Once consumable: The elements of a stream can only be consumed once during the lifetime of the stream. As stream can only have one "downstream operator". Java 8 Streams, unfortunately, do not allow general data flow graphs in which a stream from an operator is consumed by multiple operators.
- Parallelism: Operations can be executed in on data items in parallel for parallel streams. This can improve performance (but may not always).

See API Doc https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Stream<T> Interface

- Generic Interface in package `java.util.stream`.
- It provides
  - Static methods to generate stream sources (Java 8 allows static methods in interfaces)
  - Intermediate (non-terminal) operations
  - Terminal operations.

- There are specialized classes for the following three primitive types:
  - `int` → `IntegerStream`
  - `long` → `LongStream`
  - `double` → `DoubleStream`
  - There is no `BooleanStream`, no `FloatStream`, no `ShortStream`, `CharStream`, and no `ByteStream`.

  - These classes provide additional operations for numbers, e.g., min, max, sum, average.
  - Note: these classes do **not** implement `Stream<T>`, i.e., `IntStream` is not a `Stream<Integer>`.

Common Data Processing Operations in `Stream<T>`

- The following methods allow a functional programming style for data processing operations.

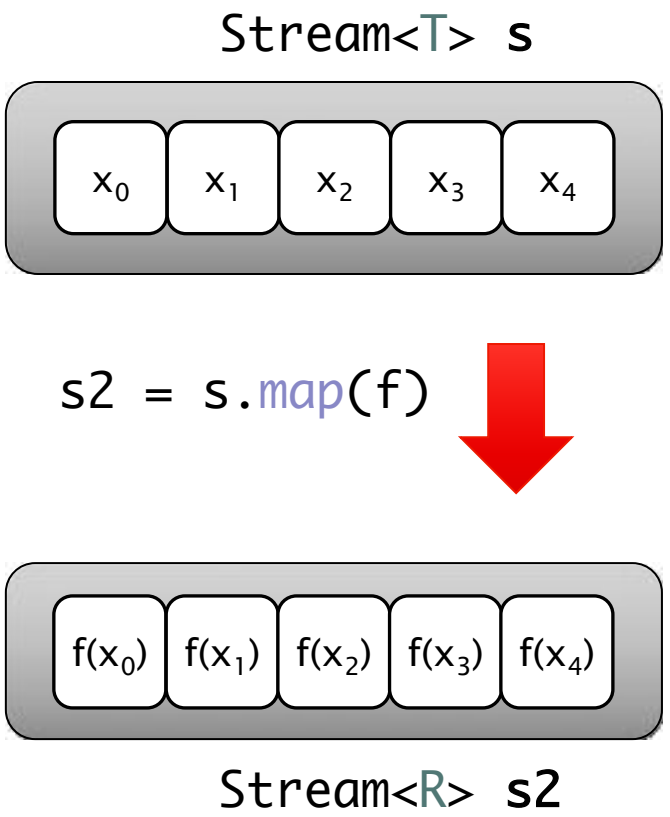
- Intermediate Operations: produce Streams
  - **map**: `Stream<T> → Stream<R>`
    Transform every element from type `T` to type `R` using a function `f: T → R`.
  - **flatMap**: `Stream<T> → Stream<R>`
    Transform every element from type `T` to a Stream of type `R` using a function `f: T → Stream<R>` then flatten out the resulting stream into one single stream.
  - **filter**: `Stream<T> → Stream<T>`
    Filter elements that satisfy a predicate `p: T → boolean`.

- Terminal Operations: produce a single value or `void`
  - **reduce**: `Stream<T> → U`
    Reduce **all** elements of type `T` to a single result value of type `R` using a reduction function.
  - **forEach**: `Stream<T> → void`
    Apply function `f: T → boolean` on every element.
  - **collect**: `Stream<T> → R`
    Performs a mutable reduction to a single value, e.g., collecting all elements as a `List<T>`.

map Operation

```
<R> Stream<R> map(Function<? super T, ? extends R> f)
```

- map(f) applies a mapper function f: T → R on every element x_i of the Stream and returns the results as anew Stream containing the elements $f(x_i)$.



- **Example:**

```
Stream<String> stream = Files.lines(Paths.get("persons.csv"));

// f: String → String
Stream<String> nameString = stream.map(line -> line.split(",")[1]);

nameString.forEach(System.out::println);
```

Output

```
Brian
Adam
Claire
Eve
Frances
Carl
Bob
Alice
Charlie
```

map Operations to Primitive Types

- Since there are specialized Streams for primitives types, there are also specialized versions of **map** that produce these Streams.

- `IntStream mapToInt(ToIntFunction<? super T> mapper);`
- `LongStream mapToLong(ToLongFunction<? super T> mapper);`
- `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);`

- **Example**: Extract age (column #3) as an `IntStream`

```
Stream<String> stream = Files.lines(Paths.get("persons.csv"));
IntStream ageStream = stream.mapToInt(line -> Integer.parseInt(line.split(",")[2]));
ageStream.forEach(System.out::println);
```

Output

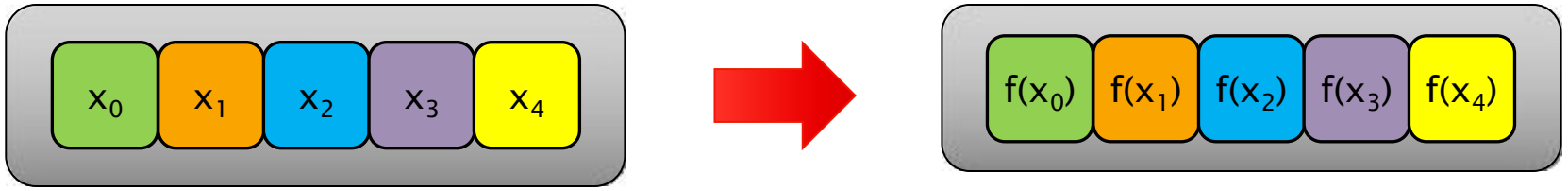
```
47
32
30
54
44
62
32
44
29
```

map and flatMap Operations

Map function: R → T
One Input Element → One Output Element

```
Stream<R> s2 = s.map(f)
```

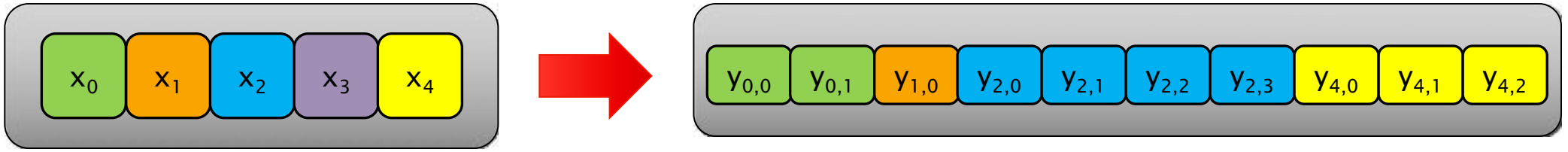
Stream<T> s

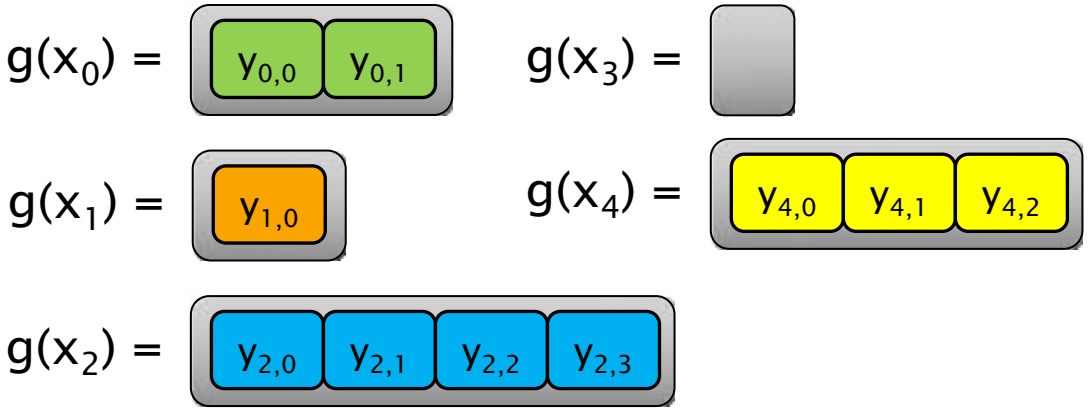


flatMap function: R → Stream<T>
One Input Element → A stream of elements (0, 1, or more)

```
Stream<R> s2 = s.flatMap(g)
```

Stream<T> s





flatMap = flatten ∘ map

1. Map elements → produces streams
2. Flatten streams into one single stream

flatMap(s) = flatten(map(s))

flatMap Example

persons.csv

```
12121,Brian,47,M
12223,Adam,32,M
12231,Claire,30,F
12326,Eve,54,F
12532,Frances,44,F
13421,Carl,62,M
12321,Bob,32,M
12512,Alice,44,F
12345,Charlie,29,M
```

Extract all columns of the **persons.csv** file as Strings.

```
Stream<String> stream = Files.lines(Paths.get("persons.csv"));


// Extract columns elements using flatMap
Stream<String> columns = stream.flatMap(line -> {
  String[] colArray = line.split(",");            // Split string into array at "," and
  Stream<String> cols = Arrays.stream(colArray);  // convert String array into Stream using
  return cols;                                    // Arrays helper class
});


// Same as above but as a one-liner.
Stream<String> columns = stream.flatMap(line -> Arrays.stream(line.split(",")));

columns.forEach(System.out::println);
```

Output

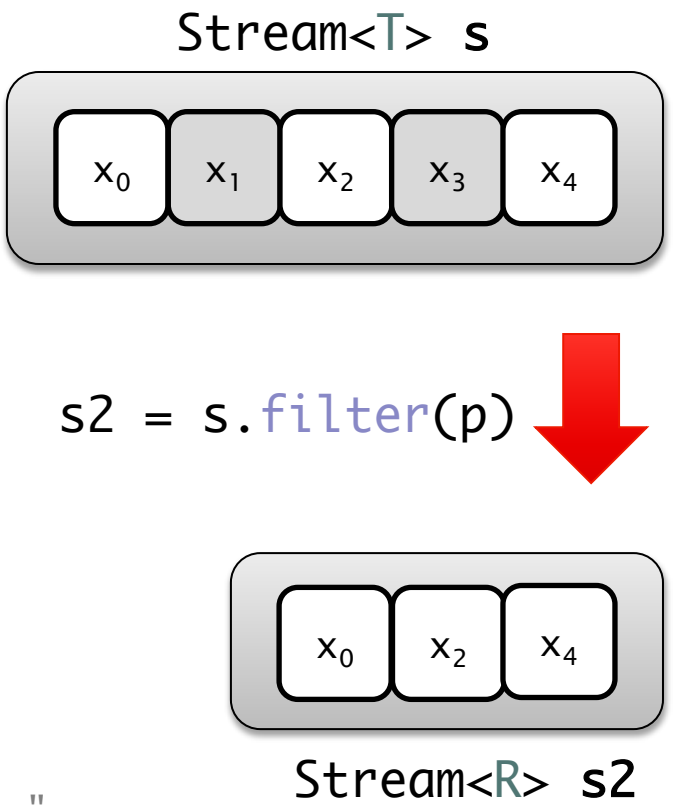
```
12121
Brian
47
M
...
12345
Charlie
29
M
```

filter Operations

`Stream<T> filter(Predicate<? super T> predicate)`

- `filter(p)` returns a stream that keeps those items x_i for which the predicate $p(x_i)$. is true.

- A Predicate is a function p: T → boolean
- Elements that satisfy the predicate are kept.



- **Example:**

```
Stream<String> stream = Files.lines(Paths.get("persons.csv"));
Stream<String> maleStream = stream
    .map(line -> line.split(","))                    // split string at ","
    .filter(cols -> cols[3].toUpperCase().equals("M"))  // filter out males
    .map(Arrays::toString);                  // convert String array to String

maleStream.forEach(System.out::println);
```

Output

```
[12121, Brian, 47, M]
[12223, Adam, 32, M]
[13421, Carl, 62, M]
[12321, Bob, 32, M]
[12345, Charlie, 29, M]
```

Sorting a Stream

`Stream<T> sorted()` ← uses natural sort order of type T

`Stream<T> sorted(Comparator<? super T> comparator)` ← sort order given by comparator

- Creates a new stream in which the elements are sorted in ascending order.
- The sort order is determined either
  - by the natural order of type T, i.e., type T implements the `Comparable<T>` interface
  - or by the specified `Comparator` object.

- Note: Sorting is a "blocking" operation, i.e., the entire stream needs to be read into memory.
  → Sorting does not work on unbounded streams.

- **Example**: Sort persons by age.

```
Stream<String> stream = Files.lines(Paths.get("persons.csv"));
Stream<String> sortedStream = stream
    .map(line -> line.split(","))          // split string at ","
    .sorted(Comparator.comparingInt(r -> Integer.parseInt(r[2])))
    .map(Arrays::toString);      // convert String array to string

sortedStream.forEach(System.out::println);
```

Parse age column (#2) as `int`, use `Comparator.comparingInt` to compare `String[]` elements extracted from the text lines.

Output

```
[12345, Charlie, 29, M]
[12231, Claire, 30, F]
[12223, Adam, 32, M]
[12321, Bob, 32, M]
[12532, Frances, 44, F]
[12512, Alice, 44, F]
[12121, Brian, 47, M]
[12326, Eve, 54, F]
[13421, Carl, 62, M]
```

Distinct Elements in Stream

```
Stream<T> distinct();
```

- Returns a stream with distinct elements as determined by the `equals()` on type `T`.

- **Example:** Distinct values in the gender column

```
Files.lines(Paths.get("persons.csv"))
    .map(line -> line.split(",")[3])
    .distinct()
    .forEach(System.out::println);
```

Output

```
M
F
```

Peeking into a Stream

```
Stream<T> peek(Consumer<? super T> action)
```

- Applies an action specified by the consumer, i.e., T → void on every element of the stream.
- Does not consume the stream, instead it returns the stream.

- This operation can be used to "inject" actions (= side effects) into the stream without consuming the stream.

- **Example:**

```
Files.lines(Paths.get("persons.csv"))
    .map(line -> line.split(","))
    .peek(fields -> System.out.println("Before filter: "
            + Arrays.toString(fields)))
    .filter(fields -> fields[3].toUpperCase().equals("M"))
    .peek(fields -> System.out.println("After filter: "
            + Arrays.toString(fields)))
    .mapToInt(fields -> Integer.parseInt(fields[2]))
    .forEach(System.out::println);
```

Output

```
Before filter: [12121, Brian, 47, M]
After filter: [12121, Brian, 47, M]
47
Before filter: [12223, Adam, 32, M]
After filter: [12223, Adam, 32, M]
32
Before filter: [12231, Claire, 30, F]
Before filter: [12326, Eve, 54, F]
Before filter: [12532, Frances, 44, F]
Before filter: [13421, Carl, 62, M]
After filter: [13421, Carl, 62, M]
62
Before filter: [12321, Bob, 32, M]
After filter: [12321, Bob, 32, M]
32
Before filter: [12512, Alice, 44, F]
Before filter: [12345, Charlie, 29, M]
After filter: [12345, Charlie, 29, M]
29
```

Trimming Streams: `skip` and `limit`

`Stream<T> skip(long n)`

- Returns a steam that is obtained by skipping the first **n** elements.
- If the stream has less than **n** elements, the resulting stream has no elements, i.e., an empty stream is returned.

- **Example**:

```
Files.lines(Paths.get("persons.csv"))
    .skip(1)     // prints only first record
    .forEach(System.out::println);
```

Output

```
12223,Adam,32,M
12231,Claire,30,F
12326,Eve,54,F
12532,Frances,44,F
13421,Carl,62,M
12321,Bob,32,M
12512,Alice,44,F
12345,Charlie,29,M
```

`Stream<T> limit(long maxSize)`

- Returns a string that contains at most `maxSize` elements.

- **Example:**

```
Files.lines(Paths.get("persons.csv"))
  .limit(3)
  .forEach(System.out::println);
```

Output

```
12121,Brian,47,M
12223,Adam,32,M
12231,Claire,30,F
```

In-class Exercise

Processing "mock" sales records

Database-like Processing of a CSV File

Example: Processing sales records in a comma-separated values (CSV) file
Download the CSV file **`mocksales.csv`** from Moodle

Header row, which we will have to skip.

CSV File

```
$ cat mocksales.csv
transaction_id,date,time,customer_id,department,amount
0,01.04.2017,8:05:57,3,Computers,94.2
1,01.04.2017,12:25:09,16,Books,46.44
2,01.04.2017,15:08:10,81,Automotive,32.51
3,01.06.2017,3:48:00,92,Clothing,88.07
4,01.06.2017,14:27:40,50,Jewelery,77.47
5,01.07.2017,3:04:44,85,Kids,21.27
6,01.07.2017,12:27:13,94,Baby,17.0
7,01.07.2017,19:17:40,29,Grocery,55.6
8,01.08.2017,12:11:50,88,Automotive,64.7
9,01.08.2017,14:12:32,55,Toys,1.89
...
```

transaction_id	date	time	customer_id	department	amount
0	01.04.2017	8:05:57	3	Computers	94.20
1	01.04.2017	12:25:09	16	Books	46.44
2	01.04.2017	15:08:10	81	Automotive	32.51
3	01.06.2017	3:48:00	92	Clothing	88.07
4	01.06.2017	14:27:40	50	Jewelery	77.47
5	01.07.2017	3:04:44	85	Kids	21.27
6	01.07.2017	12:27:13	94	Baby	17.00
7	01.07.2017	19:17:40	29	Grocery	55.60
8	01.08.2017	12:11:50	88	Automotive	64.70
9	01.08.2017	14:12:32	55	Toys	1.89

→ CSV Files can be very large (e.g., log files)
→ Do not materialize all records from file in memory!
→ Do not read in records as a list, use streams instead.

"Mock Sales"

Use Java Streams to implement the following queries:

- Query 1: Show the first 10 sales records.
- Query 2: Show all sales records for customer with ID 1.
- Query 3: Show the first 5 sales records in the "Books" department.
- Query 4: Use `sorted()` to extract the sales record with the lowest amount in the "Baby" department.
- Query 5: Use `sorted()` to extract the sales record the larges amount in the "Baby" department.

Terminal Operations

Reductions, Collections, Aggregates, forEach,
Predicates over entire Stream

Reduction Operations

- Reduction Operators consume the stream and return a single value.
- Reduction operations provided by `Stream<T>`:
  - `long count()`: Returns the number of elements in the stream.
  - `Optional<T> min(Comparator<? super T> comparator)`: Returns the smallest element using the specified comparator. If the stream is empty, the result empty, i.e., `isPresent() == false`.
  - `Optional<T> max(Comparator<? super T> comparator)`: Returns the largest element using the specified comparator. If the stream is empty, the result empty, i.e., `isPresent() == false`.

- `IntStream`, `LongStream`, and `DoubleStream` provide additional methods `sum()` and `average()`.

- General reduction operations are:
  - `T reduce(T identity, BinaryOperator<T> accumulator)`
  - `Optional<T> reduce(BinaryOperator<T> accumulator)`
  - `<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`
  - `<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)`
  - `<R, A> R collect(Collector<? super T, A, R> collector)`

reduce: Pair-wise Reduction

Performs reduction, also called a left-fold, on the stream using the associative function f.

Associative function: f(f(x, y), z) = f(x, f(y, z))

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

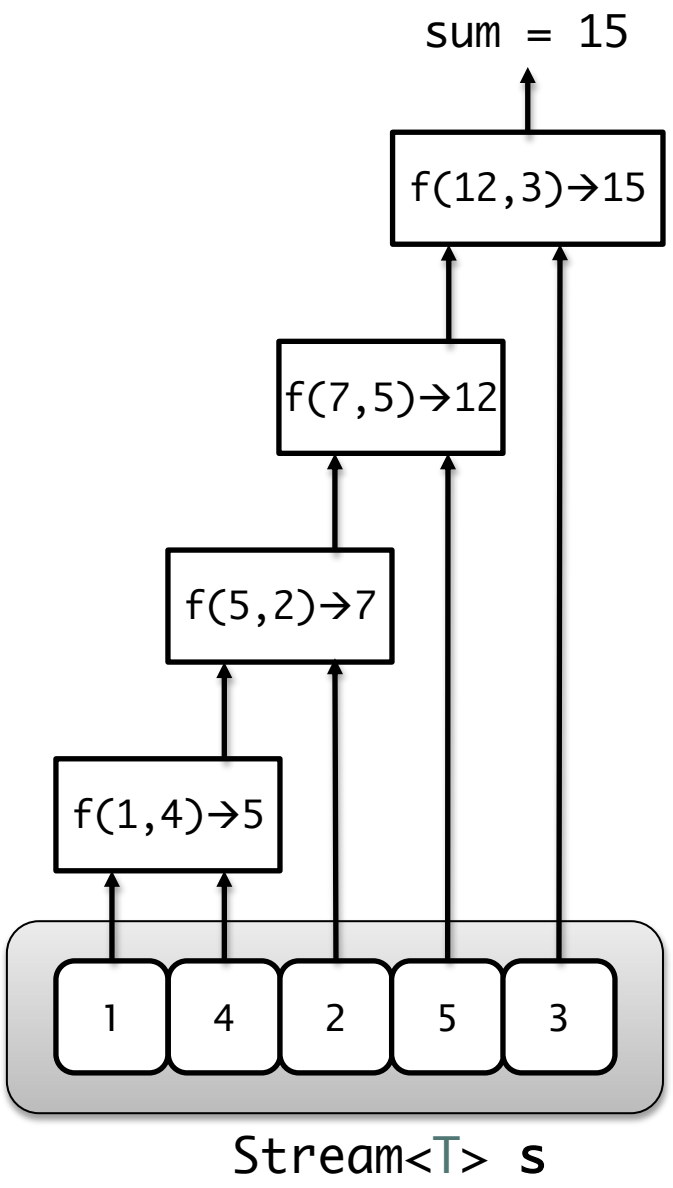
Result is an Optional. If the Stream is empty, the reduction value is not defined → Optional is empty.

Example

```
Stream<Integer> s = Stream.of(1, 4, 2, 5, 3);
```

Associative function f

```
Optional<Integer> sum = s.reduce((x, y) -> x + y);

sum.ifPresent(x -> System.out.println("sum = " + x));
```



reduce: Reduction with Identity Value

Performs reduction, also called a left-fold, on the stream using the associative function f.

```
T reduce(T identity, BinaryOperator<T> f)
```

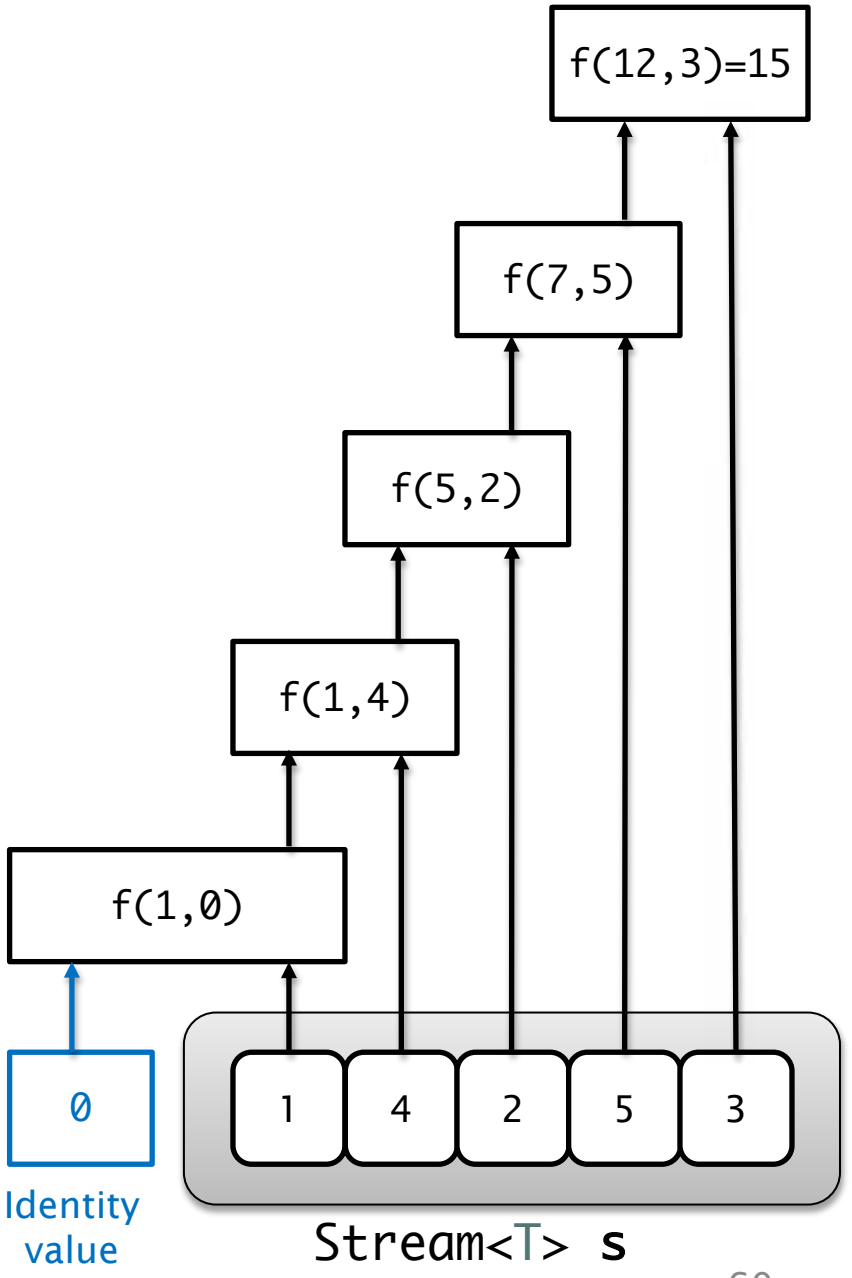
The identity value must satisfy `f(identity, t) == t`.

The result is defined even for an empty stream, thus, the result is not an Optional.

Example

```
Stream<Integer> s = Stream.of(1, 4, 2, 5, 3);

int sum = s.reduce(0, (x, y) -> x + y);

System.out.println("sum = " + sum);
```

Associative function f



reduce: Reduction with Accumulator and Combiner Functions

```
U reduce(U identity,
         BiFunction<U, ? super T, U> accumulator,  // associative function (U, T) => U
         BinaryOperator<U> combiner)               // associative function (U, U) => U
```

Example: Compute average age

Pair<S,T> class from Topic 7: first = # elements, second = sum of ages

```
Pair<Integer, Integer> aggregate = Files.lines(Paths.get("persons.csv"))
    .map(line -> Integer.parseInt(line.split(",")[2])) // extract age column
    .reduce(new Pair<>(0, 0),  // identity value
        (pair, x) -> new Pair<>(pair.getFirst() + 1, pair.getSecond() + x),  // accumulator function
        (p1, p2) -> new Pair<>(p1.getFirst() + p2.getFirst(), p1.getSecond() + p2.getSecond()));

int count = aggregate.getFirst();
int sum = aggregate.getSecond();
if (count > 0) {
  System.out.printf("average age: %.1f\n", ((double)sum)/count);
} else {
  System.out.println("count zero, average undefined");
}
```

increment count

compute new sum

Combiner function:
combines two `Pair` objects by summing up `first` and `second`.

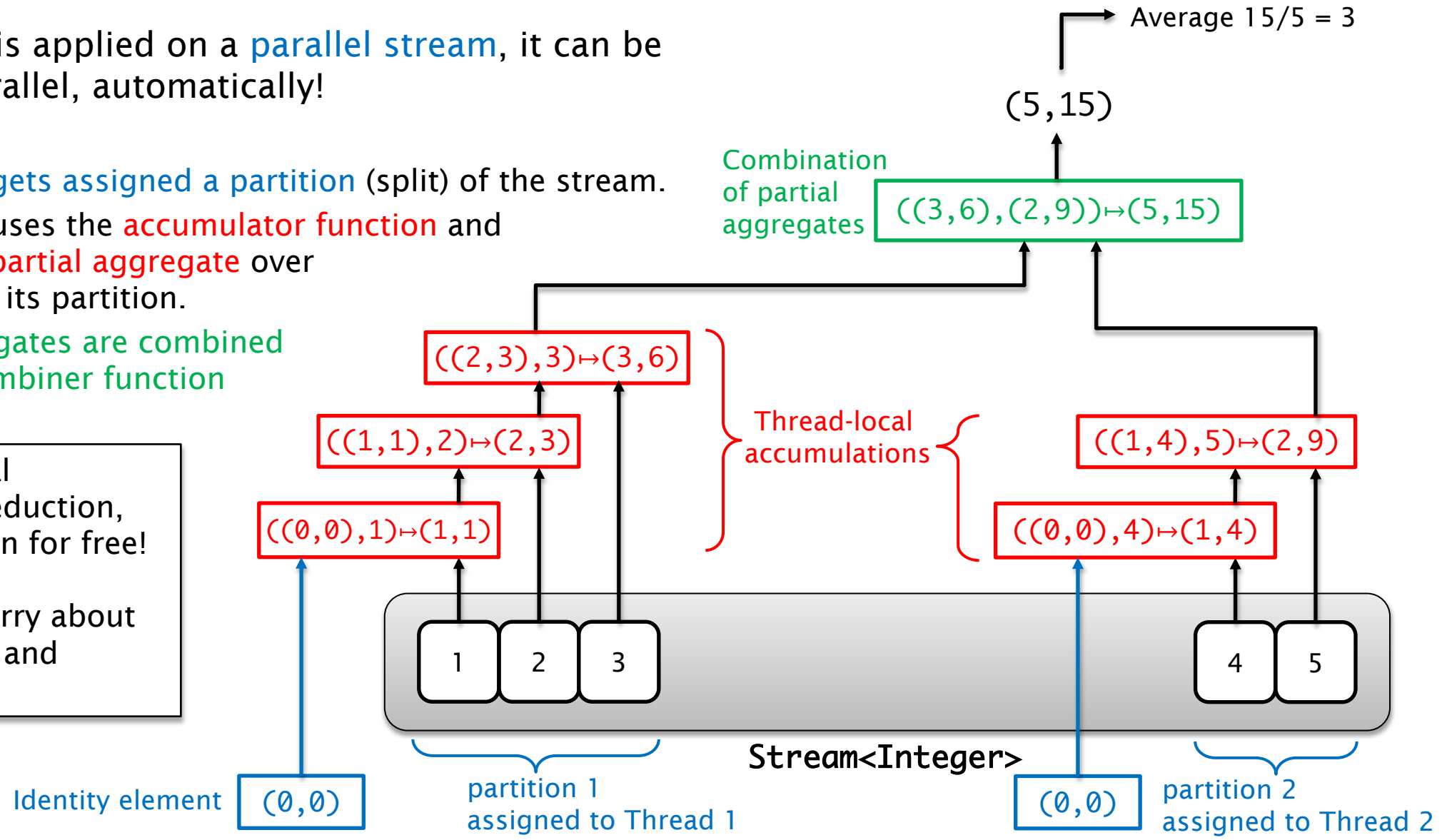
Output `average age: 41.6`

Combiner Function allows Parallel Reduction

- If the reduction is applied on a parallel stream, it can be computed in parallel, automatically!
- Execution:
  1. Each thread gets assigned a partition (split) of the stream.
  2. Each thread uses the accumulator function and computes a partial aggregate over the values of its partition.
  3. Partial aggregates are combined using the combiner function

By using a functional description of the reduction, we get parallelization for free!

We don't have to worry about threads, data races, and synchronization.



Applying Predicates on the Entire Stream

- `boolean allMatch(Predicate<? super T> predicate)`
  Returns whether all elements of this stream match the provided predicate.
  The result is the Boolean **AND** of the **`predicate`** applied all elements.

- `boolean anyMatch(Predicate<? super T> predicate)`
  Returns whether any elements of this stream match the provided predicate.
  The result is the Boolean **OR** of the **`predicate`** applied to all elements.

- `boolean noneMatch(Predicate<? super T> predicate)`
  Returns whether no elements of this stream match the provided predicate.
  The result is the Boolean **AND** of the **negated** **`predicate`** applied to all elements.

Example: Does stream only contain male persons?

```
boolean allMales = Files.lines(Paths.get("persons.csv"))
    .map(line -> line.split(",")[3])
    .allMatch(gender -> gender.toUpperCase().equals("M"));
System.out.println("all males: " + allMales);
```

Returning a Single Element from a Stream

`Optional<T> findFirst()`

- Returns an optional yielding the first element in the stream.
- If the stream is empty, the optional evaluates to empty.
- If the stream is unordered, any element may be returned.

`Optional<T> findAny()`

- Returns an optional yielding any element from the stream.
- If the stream is empty, the optional evaluates to empty.
- This operation is explicitly non-deterministic.

Example:

```
Optional<String> firstPerson = Files.lines(Paths.get("persons.csv")).findFirst();
Optional<String> anyPerson = Files.lines(Paths.get("persons.csv")).findAny();

firstPerson.ifPresent(p -> System.out.println("first: " + p));
anyPerson.ifPresent(p -> System.out.println("any: " + p));
```

Output

```
first: 12121,Brian,47,M
any: 12121,Brian,47,M
```

Collecting Elements from a Streams using Collectors

- Collect all elements into an array:

```
Object[] toArray()
A[] toArray(IntFunction<A[]> generator)
```

Function that takes an `int` **n** and creates an array `A[]` of length **n**.

- Collect all elements into an object of Type `R` using an instance of the interface `Collector<T, A, R>`:

```
R collect(Collector<? super T, A, R> collector)
```

  You need to implement a number of methods:
  - `Supplier<A> supplier()`: returns function that generates new mutable result container.
  - `BiConsumer<A, T> accumulator()`: returns function that adds new element to container.
  - `BinaryOperator<A> combiner()`: returns combiner function that combines two containers.
  - `Function<A, R> finisher()`: returns finisher function that converts container into final result type

- The most commonly used collectors are implemented in the helper class `Collectors`:
  - `Collector<T, ?, List<T>> toList()`
  - `Collector<T, ?, Set<T>> toSet()`
  - `Collector<CharSequence, ?, String> joining()`
  - `Collector<CharSequence, ?, String> joining(CharSequence delimiter)`

Collecting Elements: Example

- Collecting names as String array:

```
String[] names = Files.lines(Paths.get("persons.csv"))
    .map(line -> line.split(",")[1])
    .toArray(String[]::new);  // Reference to array constructor is a matching generator
for (String name : names) {
  System.out.println(name);
}
```

- Collecting ages as a List<Integer>:

```
List<Integer> ages = Files.lines(Paths.get("persons.csv"))
    .map(line -> Integer.parseInt(line.split(",")[2]))
    .collect(Collectors.toList());
for (int age : ages) {
  System.out.println(age);
}
```

- Collecting names as single String joined by ", ":

```
String nameStr = Files.lines(Paths.get("persons.csv"))
    .map(line -> line.split(",")[1])
    .collect(Collectors.joining(", "));
System.out.println(nameStr);
```

Output

```
Brian, Adam, Claire, Eve, Frances, Carl, Bob, Alice, Charlie
```

forEach

```
void forEach(Consumer<? super T> action)
```

- Performs action for each element in the stream.

```
public interface Consumer<T> {
  void accept(T t);
}
```

- This operation is similar to the intermediate operation `peek(action)`, however, `forEach(action)` terminates the stream.
- This operation is used for side-effects.
- When executed on parallel streams, there are no guarantees provided with respect to the order of the elements.

- **Example:**

```
Stream<String> columns = stream.flatMap(line -> Arrays.stream(line.split(",")));
columns.forEach(System.out::println);  // action specified as a method reference
```

Exercise 5: Problem 2

More Queries on the "Mock Sales" Dataset

Stream Sources

Streams from Arrays and Collections, Stream Builder, Random Number Generator, Files

Streams from Collections and Arrays

- Use the generic static method `stream(T[] array)` from helper Class `Arrays`:

```
String[] stringArray = {"Alice", "Bob", "Eve", "Gaby" };
Stream<String> stream = Arrays.stream(stringArray);
```

- For generating streams from individual values you can use `Streams.of(T... values)` which accepts a variable argument list:

```
Stream<Integer> numberStream = Stream.of(1, 2, 3, 4, 5, 6);
Stream<String> stringStream = Stream.of("Alice", "Bob", "Eve", "Gaby");
```

Custom Stream Generators: Seed + Update

- For ranges of values or sequences with a particular step pattern use
  `Stream<T> iterate(final T seed, final UnaryOperator<T> f):`

- **seed** is the value of the first element produced in the stream.
- The function f is the update function f: T → T

- **Example**: Infinite Stream of positive odd numbers:

```
Stream<Integer> oddStream = Stream.iterate(1, i -> i + 2);
System.out.println(oddStream
      .limit(10)
      .map(Object::toString)
      .collect(Collectors.joining(", ")));
```

Output

```
1, 3, 5, 7, 9, 11, 13, 15, 17, 19
```

Custom Stream Generators: Supplier

- You can also provide an implementation of a `Supplier<T>` that produces new elements for an infinite stream. To the static method `generate(..)` of `Stream<T>`.

```
static <T> Stream<T> generate(Supplier<T> s)
```

with

```
public interface Supplier<T> {
  T get();
}
```

- **Example:** Fibonacci Sequence Stream

```
Stream<Integer> fibonnaciStream = Stream.generate(new Supplier<Integer>() {
  int next = 1;
  int last = 0;

  @Override
  public Integer get() {
    int result = next;
    next += last;
    last = result;
    return result;
  }
});
System.out.println(fibonnaciStream.limit(10).map(Object::toString)
                                  .collect(Collectors.joining(", ")));
```

Output

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Stream Builder

- A stream builder can be used to create the a stream element-by-element without the overhead of having to create an array as a temporary array.

```
static <T> Builder<T> builder()
```

  - Add elements using the method `add(T t)`. You can chain the methods (Fluent Style)
  - Finalize the stream by calling the `build()` method.

- **Example:**

Here you need to provide a **type witness** for the static generic method `builder()`, otherwise, the compiler cannot infer the value of the type parameter.

```
Stream<String> stream = Stream.<String>builder().add("Alice").add("Bob").add("Charlie").build();
System.out.println(stream.collect(Collectors.joining(", ")));
```

Output

```
Alice, Bob, Charlie
```

Streams from a Random Number Generator

- Generate a Stream from a `java.util.Random` pseudo random number generator:
  - `IntStream ints(int minValue, int maxValueExclusive)`:
    Generates uniform distributed integer in [`minValue`, `maxValueExclusive`).
  - `DoubleStream doubles(int minValue, int maxValueExclusive)`:
    Generates uniform distributed double in [`minValue`, `maxValueExclusive`).

- **Example**: Random fair die rolls and fair coin tosses

```
Random random = new Random();
IntStream dieRollsStream = random.ints(1, 7);
Stream<Boolean> coinHeadsStream = random.doubles(0, 1).mapToObj(x -> x > 0.5);

System.out.println(dieRollsStream
    .mapToObj(Integer::toString)
    .limit(10)
    .collect(Collectors.joining(", ")));

System.out.println(coinHeadsStream
    .map(head -> head ? "head" : "tail")
    .limit(10)
    .collect(Collectors.joining(", ")));
```

Output

```
1, 5, 2, 1, 2, 4, 2, 3, 2, 2
head, head, tail, head, tail, head, tail, tail, head, tail
```

Streams from Files

- Extract Text Lines from Text File:

```
class Files { ...
  static Stream<String> lines(Path path) throws IOException
}
```

Example:

```
Files.lines(Paths.get("persons.csv"))
    .forEach(System.out::println);
```

- Extract entries from Zip or Jar file (metadata of entries):

Example:

```
JarFile jarFile = new JarFile("lib/kotlin-stdlib.jar");
jarFile.stream()
    .map(ZipEntry::getName)
    .filter(name -> name.contains("Pair"))
    .findFirst().ifPresent(System.out::println);
```

Output `kotlin/Pair.class`

Streams from Directories

- Entries in current directory, does not descend recursively into subdirectories:

Example:

```
Files.list(Paths.get("."))
    .forEach(System.out::println);
```

Output

```
./english_stopwords_nltk.txt
./oop2-code.iml
./out
./alice_in_wonderland.txt
./Pair_disass.txt
./persons.csv
./cpp
./lib
./mocksales.csv
./.idea
./src
```

- Walk directory tree recursively and return encountered entries as stream:

Example:

```
Files.walk(Paths.get("."))
    .limit(8)
    .forEach(System.out::println);
```

Output

```
.
./english_stopwords_nltk.txt
./oop2-code.iml
./out
./out/production
./out/production/oop2-code
./out/production/oop2-code/ch
./out/production/oop2-code/ch/bfh
```

Parallel Streams

Executing Streams in Parallel

- Streams can be data parallel, i.e., different elements can be processed in parallel.
- The elements in a parallel stream are split into chunks. Each chunk is potentially processed by a different thread.
- Parallel processing of stream is most effective on reduction operations.
  The aggregate or condense many data items into one (high fan-in, low fan-out).

- Using functional operations like **`map`**, **`filter`**, and **`reduce`** with side-effect free lambdas, the computations are automatically executed parallel.
  This is one of major powers of functional programming is so important!

- Parallel streams are executed using Java's default `ForkJoinPool` (*we have not discussed that*).
  - Tasks submitted to this pool can be split recursively into smaller sub tasks that are executed concurrently.
  - The number of CPUs cores to assigned to this pool can be controlled using
    ```
    System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "8");
    ```
  - By default, the property is set equal to the number of available logical cores in the system.

Creating Parallel Streams

From a `Collection<E> c`:

`c.parallelStream()` instead of `c.` `c.stream()` From

- Note that both methods return a `Stream<E>`.
- Use the `isParallel()` method to check whether a stream is parallel.

```
Random rnd = new Random();
List<Integer> numbersList = new ArrayList<>();
for (int i = 0; i < 1_000_000; i++) {
  numbersList.add(rnd.nextInt(1001));
}
long sum = numbersList.parallelStream()
    .mapToLong(Integer::intValue)    // convert Stream<Integer> to IntStream
    .reduce(0L, Long::sum);
```

From a non-parallel Stream `s`:

use `s.parallel()` method.

- For this to work, the stream must be splittable, for example, it must not be infinite.

```
long sum = Stream.iterate(1L, i -> i + 1)
    .limit(1_000_000)
    .parallel()
    .reduce(0L, Long::sum);
```

Caveats for Parallel Streams

- Streams must be splittable, i.e., they must be finite.
- The elements in parallel streams are not ordered.
  - Don't rely on order.
  - You can use `forEachOrdered(..)` on the stream but this almost always defeats the performance advantage of using parallel execution.
- Lambda functions must not have side effects.
  - Avoid stateful lambda expression.
  - Unforeseen data races may otherwise occur when the lambda is executed by multiple threads.

```
private static int count = 0;
private static void foo() {
  long sum = Stream.iterate(1L, i -> i + 1)
      .limit(1_000_000)
      .parallel()
      .map(i -> {
        count++;    // side effect with data race (DOUBLY BAD)!
        return 1L;
      })
      .reduce(0L, Long::sum);
  System.out.println("sum: " + sum);
  System.out.println("count: " + count);
}
```

Output

```
sum: 1000000
count: 230484
```

Exercise 5: Problem 3

Parallel Walk of the Directory Tree using Streams

Summary

Java 8 Language Features

- Functional Interfaces: Interfaces with exactly one abstract method. A matching lambda expression can be regarded as implementation of a functional interface. Hack to provide signatures for higher-order functions.
- Method References: Passing a reference to a function reference as a lambda.
- `Optional<T>` Monad: Container that either has a value or is empty. Allows exception free programming as they can be freely passed around.

Java 8 Streams

- Stream represents a sequence of elements.
- Elements are processed in a pipeline of operations.
- Intermediate operations apply a function onto stream elements and produce another stream with results.
- Terminal operation end the stream in a single value or `void`.
- Intermediate operation: map, flatMap, filter, sort, distinct, peek, skip, limit
- Terminal operation: reductions (folds), collect elements into collections, predicates on entire streams.
- Operations map, filter, and reduce are primitives that can be used to operate on data in a functional manner.
- Streams using operators that are side-effect free can be parallelized for free (data-parallel not operator-parallel).