7SENG011W Object Oriented Programming

Collections: Lists and Maps; Sorting; Overview of Streams and Text Files

Dr Francesco Tusa

Readings

Books

- Head First Java
  - Chapter 11. Data Structures: Collections and Generics
  - Chapter 16. Saving Objects (and Text): Serialization and File I/O

Online

- Java Documentation: Collections Framework Overview
- The Java Tutorials: Collections
- Javadoc: Comparator Interface and Comparable Interface
- Java Documentation: I/O Streams

Outline

- Java Collections Framework
  - Introduction
  - Lists
  - Maps
- Sorting Collections
  - Comparator interface
  - Comparable interface
- Streams and Files
  - Standard Streams and System class
  - Writing to and Reading text data from a File
  - try-with-resources

Java Collections Framework

- For many applications, you want to create and manage **groups of related objects**

- **Arrays** are most useful for creating and working with a **fixed number** of objects

```
Shape[] shapes = new Shape[5];
```

Java Collections Framework

- For many applications, you want to create and manage **groups of related objects**

- **Arrays** are most useful for creating and working with a fixed number of objects

- What if we need a **data structure** that can **grow** and **shrink dynamically** as the needs of the application change?

Java Collections Framework

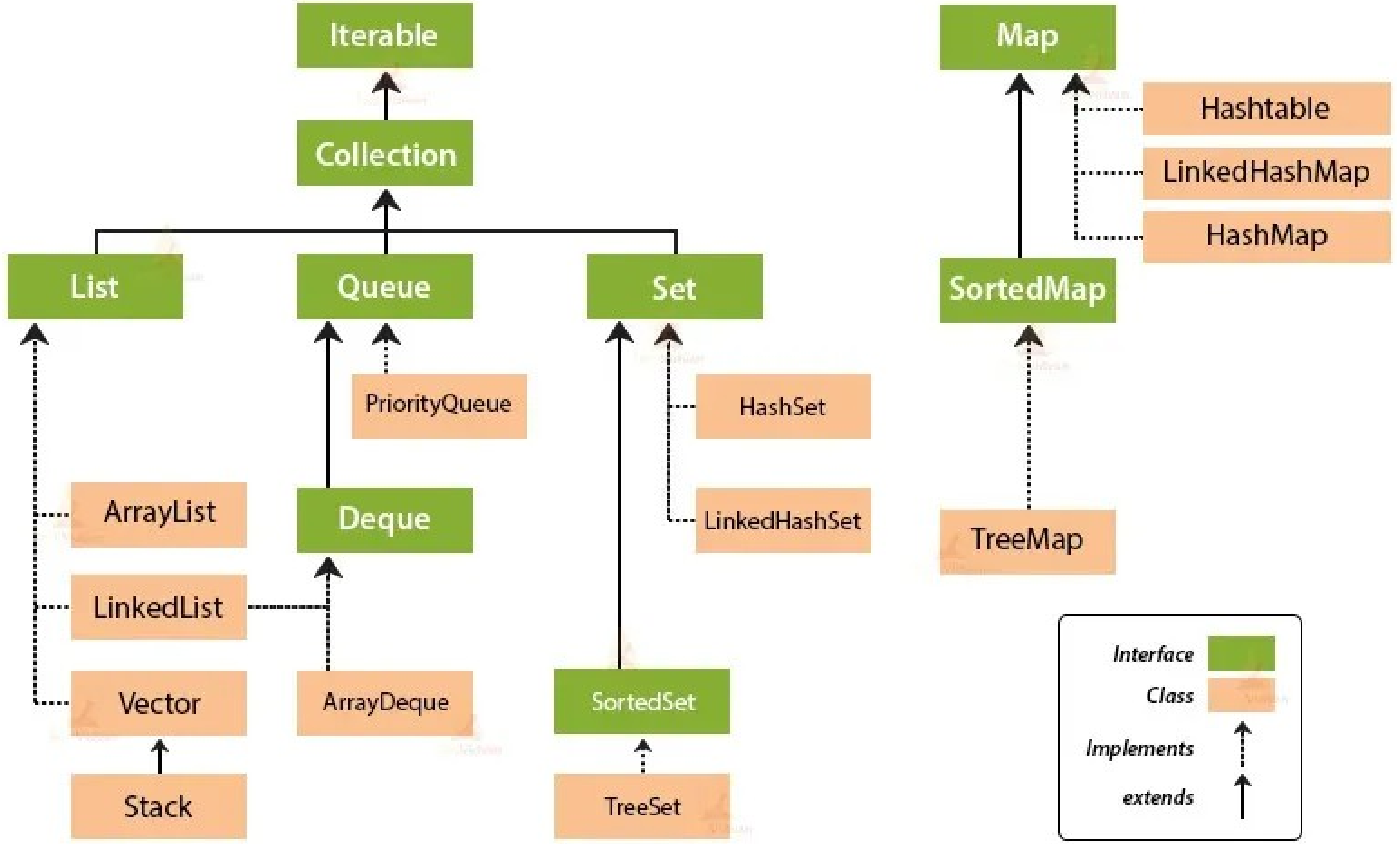
The ***java.utils*** *package* contains interfaces and classes to define and manipulate various **collections** of *objects*:

- **List, Map** (today)
- **Set, Queue**

- **Arrays** are also collections and can **interwork** with the framework: conversion from/to **List**

Java Collections Framework

The ***java.utils*** *package* contains interfaces and classes to define and manipulate various **collections** of *objects*:

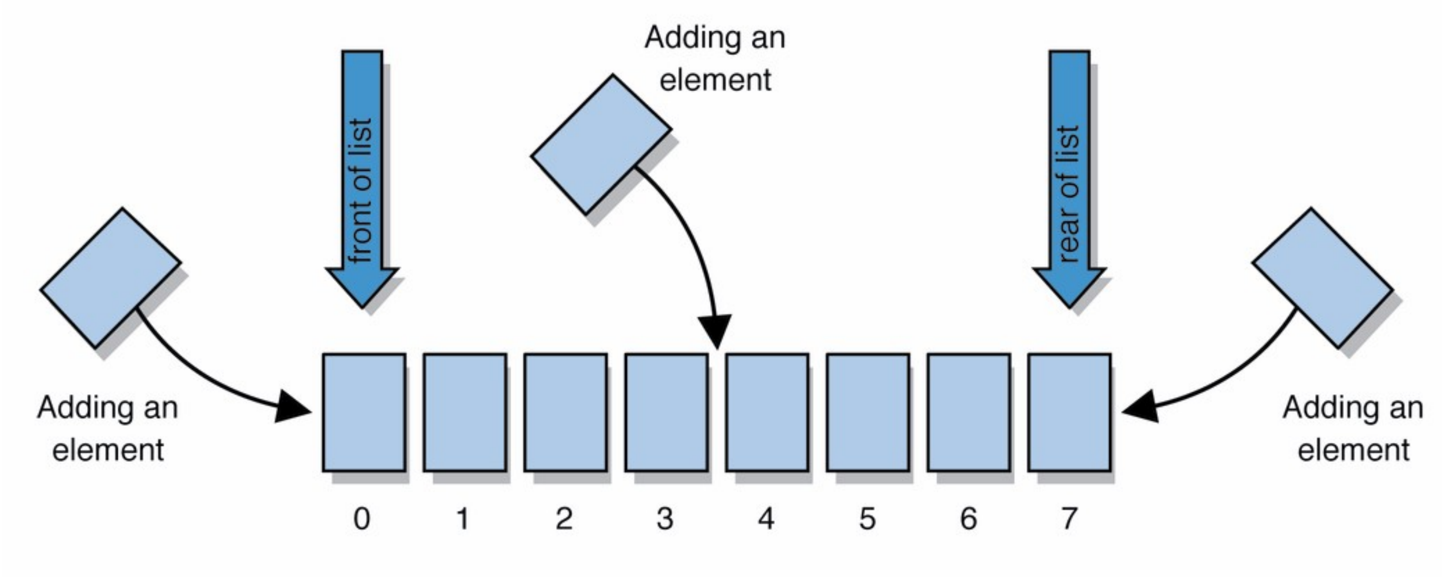


Outline

- Java Collections Framework
  - Introduction
  - **Lists**
  - Maps
- Sorting Collections
  - Comparator interface
  - Comparable interface
- Streams and Files
  - Standard Streams and System class
  - Writing to and Reading text data from a File
  - try-with-resources

Java Collections Framework: List

A **list** contains an **ordered variable-length** *sequence* of elements that can be individually **added** and **accessed** by index.



Java Collections Framework: interface *List*

Defines a **contract** that every **list** implementation should fulfil.

Java Collections Framework: interface *List*

- **add**(element): *adds element* at the end of the list
- **add**(index, element): *adds element* in the position *index* of the list
- **get**(index): returns the *element* in the position *index* in the list
- **remove**(index): *removes* an element at a particular *index*
- **contains**(element): determines whether *element* is in the list
- **indexOf**(element): returns the *index* of the first occurrence of *element*
- **size**(): *returns* the *number of elements* of the list

Java Collections Framework: *ArrayList*

- Implements the *List* interface using an **array** dynamically **reallocated** as required by the **list size**
- Alternative implementation: *LinkedList*

How do we specify the type of elements of the list?

Generics Collections: *ArrayList*<*type*>

List<type> list = new ArrayList<type>();

- **notation** with <*type*>
- **Strictly Typed**: can only contain objects of a specified *type* and *prevents* the addition of *incompatible* objects.
- **Type Safety**: the type of elements in the collection is specified at compile time, reducing runtime errors.
- Next **example:**

List<Student> list = new ArrayList<Student>();

ArrayList<*type*>: *Student* code example

List<*type*> is an *interface* whose elements are of *type* Student

```
{
    List<Student> students = new ArrayList<Student>();
    Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
    Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
    Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

    students.add(s1);
    students.add(s2);
    students.add(s3);

    students.remove(0);
    students.add(0, new Teacher("Emily", "Johnson", 1980, 35000, "Math"));


    for (int i=0; i < students.size(); i++)
    {
      System.out.println(students.get(i).getFee());
    }
}
```

i-th student object from the list

ArrayList<type>: Student code example

ArrayList<*type*> is a list implementation that uses *arrays* internally, and whose elements are of *type* Student

```
{
    List<Student> students = new ArrayList<Student>();
    Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
    Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
    Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

    students.add(s1);
    students.add(s2);
    students.add(s3);

    students.remove(0);
    students.add(0, new Teacher("Emily", "Johnson", 1980, 35000, "Math"));


    for (int i=0; i < students.size(); i++)
    {
       System.out.println(students.get(i).getFee());
    }
}
```

i-th student object from the list

ArrayList<*type*>: *Student* code example

The declaration can be simplified <>: the type is the same as the one of *List<Student>*

```
{
      List<Student> students = new ArrayList<>();
      Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
      Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
      Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

      students.add(s1);
      students.add(s2);
      students.add(s3);

      students.remove(0);
      students.add(0, new Teacher("Emily", "Johnson", 1980, 35000, "Math"));


      for (int i=0; i < students.size(); i++)
      {
         System.out.println(students.get(i).getFee());
      }
}
```

i-th student object from the list

ArrayList<type>: Student code example

```
{
        List<Student> students = new ArrayList<>();
        Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
        Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
        Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

        students.add(s1);
        students.add(s2);
        students.add(s3);

        students.remove(0);
        students.add(0, new Teacher("Emily", "Johnson", 1980, 35000, "Math"));


        for (int i=0; i < students.size(); i++)
        {
            System.out.println(students.get(i).getFee());
        }
}
```

Will this compile?

ArrayList<type>: Student code example

```
{
    List<Student> students = new ArrayList<>();
    Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
    Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
    Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

    students.add(s1);
    students.add(s2);
    students.add(s3);

    students.remove(0);
    students.add(0, new Teacher("Emily", "Johnson", 1980, 35000, "Math"));


    for (int i=0; i < students.size(); i++)
    {
       System.out.println(students.get(i).getFee());
    }
}
```

No, *students* is a **strongly typed** *list*. **Only** references to *Student* objects can be added

ArrayList<*type*>*:* Search for an Element

How can we search for a student with a given *studentNumber* (or *surname*) in the *students* list?

```
List<Student> students = new ArrayList<>();

students.add(new Student("Michael", "Johnson", 1998, 54321, 4800));
students.add(new Student("William", "Taylor", 1999, 35791, 6000));
students.add(new Student("Elisabeth", "Smith", 1995, 12345, 5000));

...

students.add(new Student("John", "Doe", 2001, 45678, 5600));
```

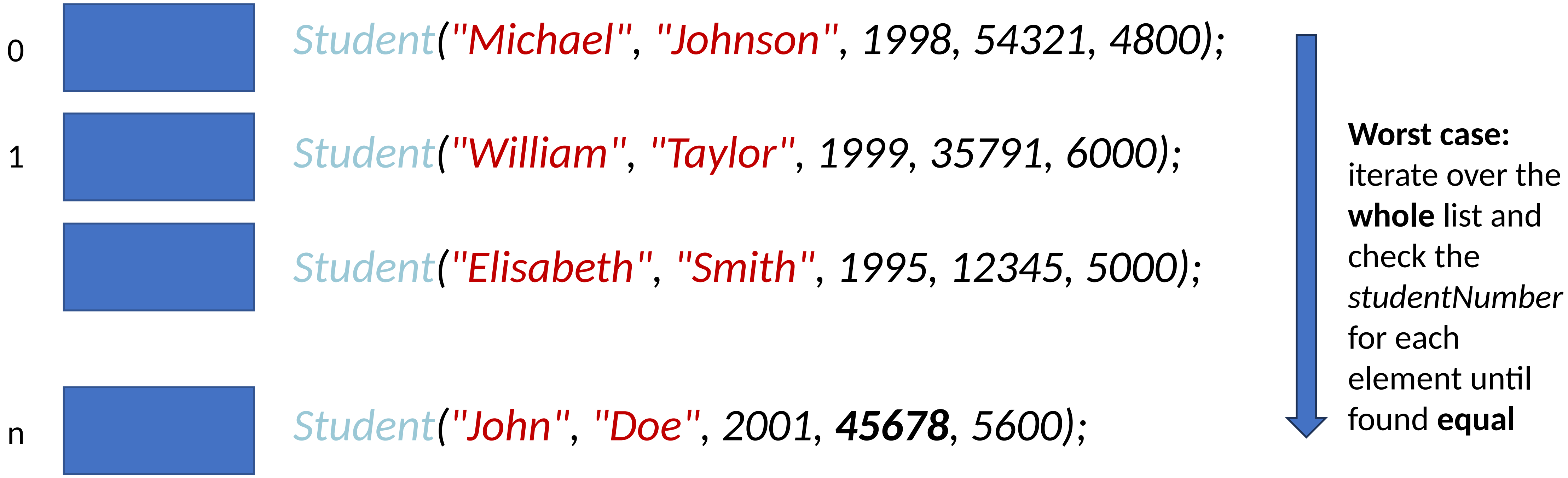
ArrayList<*type*>*:* Search for an Element

For example: *studentNumber* equals ***45678***?

```
List<Student> students = new ArrayList<>();

students.add(new Student("Michael", "Johnson", 1998, 54321, 4800));
students.add(new Student("William", "Taylor", 1999, 35791, 6000));
students.add(new Student("Elisabeth", "Smith", 1995, 12345, 5000));

...

students.add(new Student("John", "Doe", 2001, 45678, 5600));
```

ArrayList<*type*>*:* Search for an Element

For example: *studentNumber* equals ***45678****?*



Student("Michael", "Johnson", 1998, 54321, 4800);

Student("William", "Taylor", 1999, 35791, 6000);

Student("Elisabeth", "Smith", 1995, 12345, 5000);

*Student("John", "Doe", 2001, **45678**, 5600);*

Worst case: iterate over the **whole** list and check the *studentNumber* for each element until found **equal**

Question

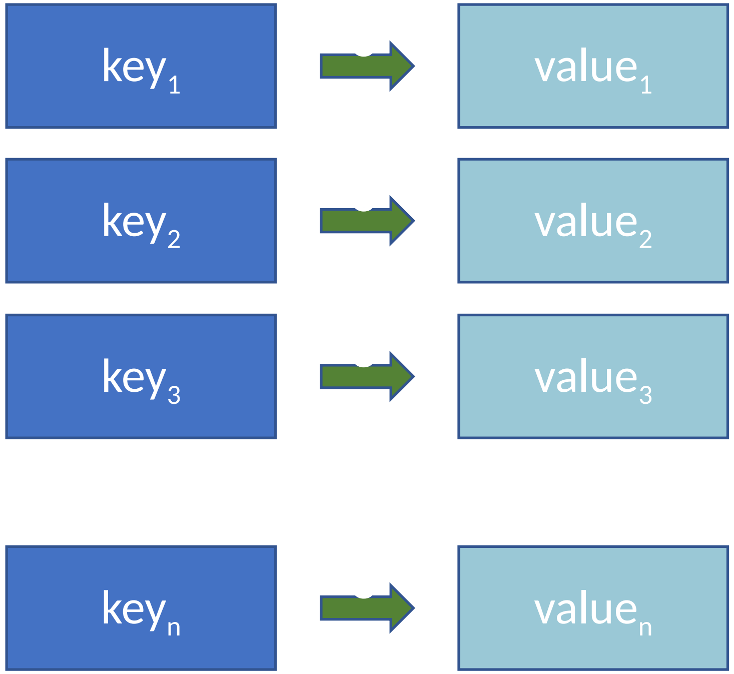
- Is this **efficient** if the list is **large**?
- What if we need to **remove** the element in **position 0**?
- **Computational Complexity** (more in Algorithms and Data Structures Module)

Outline

- Java Collections Framework
  - Introduction
  - Lists
  - **Maps**
- Sorting Collections
  - Comparator interface
  - Comparable interface
- Streams and Files
  - Standard Streams and System class
  - Writing to and Reading text data from a File
  - try-with-resources

Map Collection

- A **map** represents a collection of **key-value pairs**
- Each *key* is **unique** and can be used to **look up** a *value*



Examples:

- Phone Book
- OS file extensions to default applications mapping

Java Collections: interface *Map*<K, V>

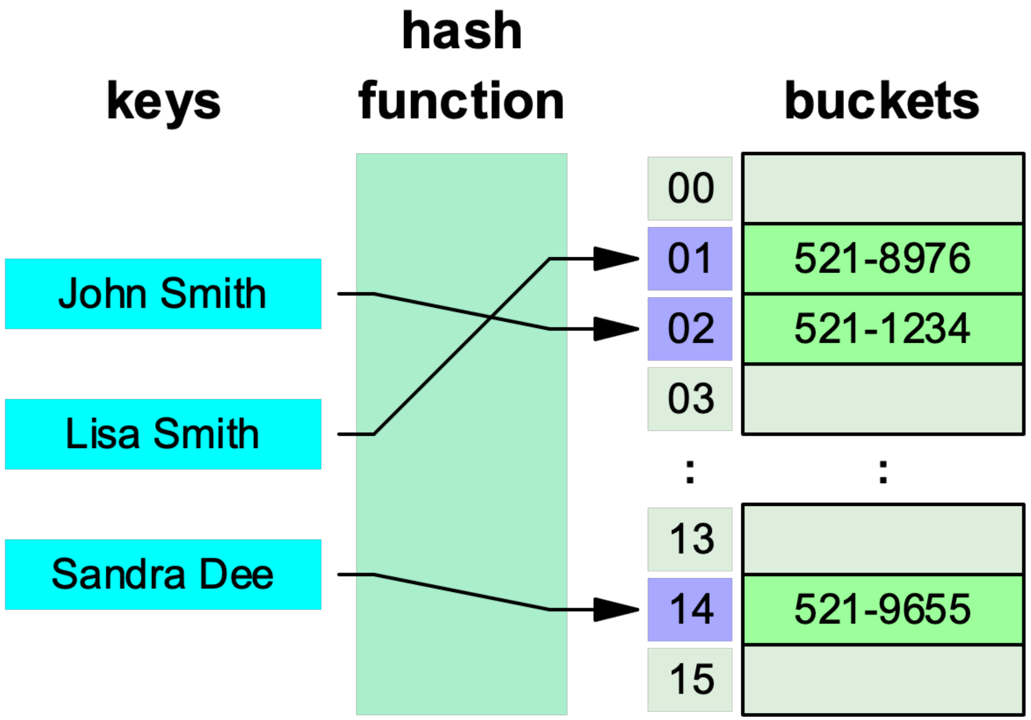
Map<K, V> Defines a **contract** that every implementation of **map** of *keys* of type *K* and *values* of type *V* should fulfil.

Java Collections: interface *Map*<K, V>

- **put**(key, value): *adds* the specified *key* and *value* to the map
- **get**(key): *get* the value *associated* with the specified *key*
- **remove**(key): removes the mapping for a *key* from this map if it is present
- **containsKey**(key): determines whether the map *contains* the specified *key*
- **size**(): returns the number of key-value mappings
- **values**(): returns a *Collection* view of the values contained in this map

Java Collections Framework: *HashMap*

- Implements the *Map*<*K*, *V*> interface using a **Hash Table**
- It is like an array where the *index* is determined by a **hash** of the **key**
- It **does not** guarantee any specific **order** of elements.



Java Collections Framework: *HashMap*

- Implements the *Map<K, V>* interface using a **Hash Table**
- It is like an array where the *index* is determined by a **hash** of the **key**
- It **does not** guarantee any specific **order** of elements.

- Alternative implementations: *LinkedHashMap* and *TreeMap* allow for **the sorting** of entries based on *insertion* order or *key* order.

- Next **example**:

```
Map<Integer, Student> students = new HashMap<>();
```

HashMap<*Integer*, *Student*> code example

```
{
  Map <Integer, Student> students = new HashMap<>();









}
```

Integer is a **wrapper class** that encapsulates an *int* value in an object, providing methods for converting and manipulating integers.

Primitive types **cannot be used directly** with *Lists* and *Maps*

HashMap<Integer, Student> code example

key-value pairs are added

```
{
   Map <Integer, Student> students = new HashMap<>();

   students.put(12345, new Student("Elisabeth", "Smith", 1995, 12345, 5000));
   students.put(54321, new Student("Michael", "Johnson", 1998, 54321, 4800));
   students.put(35791, new Student("William", "Taylor", 1999, 35791, 6000));
```

key: Integer

value: Student

```
}
```

HashMap<Integer, Student> code example

```
{
    Map <Integer, Student> students = new HashMap<>();

    students.put(12345, new Student("Elisabeth", "Smith", 1995, 12345, 5000));
    students.put(54321, new Student("Michael", "Johnson", 1998, 54321, 4800));
    students.put(35791, new Student("William", "Taylor", 1999, 35791, 6000));

    System.out.println(students.get(54321).getFee()); // accesses the value corresponding to key 54321

    students.remove(35791); // removes the key and associated value
    System.out.println(students.size()); // prints 2

    for (Student s : students.values())  // iterates over the map values
    {
        System.out.println(s.getYearOfBirth()); // calls getter for yearOfBirth on each value
    }
}
```

Other Generics Collections

- **Set<T>**: contains *no duplicate* elements stored in *no particular order*
- **Queue<T>**: like a list that supports *First In First Out* (FIFO)
- **Stack<T>**: like a list that supports *Last In First Out* (LIFO)

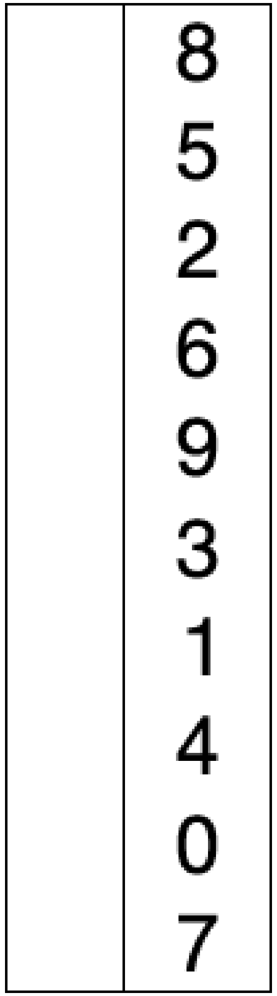
(more in Algorithms and Data Structures Module)

Outline

- Java Collections Framework
  - Introduction
  - Lists
  - Maps
- Sorting Collections
  - **Comparator interface**
  - Comparable interface
- Streams and Files
  - Standard Streams and System class
  - Writing to and Reading text data from a File
  - try-with-resources

Sorting problem and algorithms

- A sorting algorithm **arranges elements** of a list **into an order**.
- **Numerical** order and **lexicographical** order, either *ascending* or *descending*.



	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Red is current min.
Yellow is sorted list.
Blue is current item.

Sorting problem and algorithms

- A sorting algorithm **arranges elements** of a list **into an order**.
- **Numerical** order and **lexicographical** order, either *ascending* or *descending*.

- A mechanism to **compare** the elements is required: <, > or = for *numbers:*

  3 < 9 means 3 **precedes** 9 when using *numerical ascending ordering*

Sorting problem and algorithms

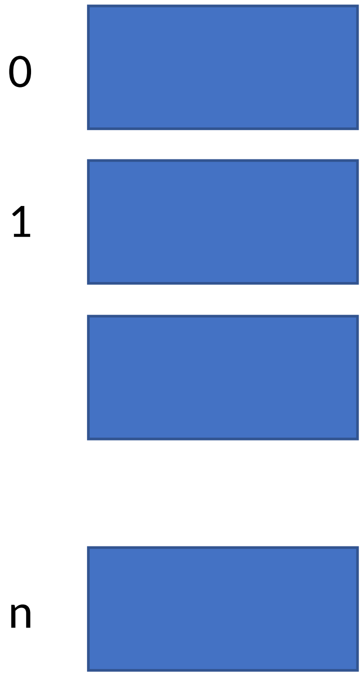
- A sorting algorithm **arranges elements** of a list **into an order**.
- **Numerical** order and **lexicographical** order, either *ascending* or *descending*.

- A mechanism to **compare** the elements is required: <, > or = for *numbers*:

  3 < 9 means 3 **precedes** 9 when using *numerical ascending ordering*

What happens if the elements are ***not simply numbers****?*

Sorting problem and algorithms



```
Student("Michael", "Johnson", 1998, 54321, 4800);
Student("William", "Taylor", 1999, 35791, 6000);
Student("Elisabeth", "Smith", 1995, 12345, 5000);
Student("John", "Doe", 2001, 45678, 5600);
```

Sorting problem and algorithms

We cannot simply **compare** *s1* and *s2* with **relational operators** <, > or =

```
Student s1 = Student("Michael", "Johnson", 1998, 54321, 4800);

Student s2 = Student("William", "Taylor", 1999, 35791, 6000);
```

We need to define a **custom mechanism** to **compare** two students.

Sorting problem and algorithms

We cannot simply **compare** *s1* and *s2* with **relational operators** <, > or =

```
Student s1 = Student("Michael", "Johnson", 1998, 54321, 4800);

Student s2 = Student("William", "Taylor", 1999, 35791, 6000);
```

Given two *Student* objects, according to **what criterion** is a given student ***less than***, ***greater than*** or ***equal to*** another?

Sorting problem and algorithms

We cannot simply **compare** *s1* and *s2* with **relational operators** <, > or =

```
Student s1 = Student("Michael", "Johnson", 1998, 54321, 4800);

Student s2 = Student("William", "Taylor", 1999, 35791, 6000);
```

Possible alternatives: *name*, *surname*, *year of birth*, *student number* and *fee*

Sorting problem and algorithms

We cannot simply **compare** *s1* and *s2* with **relational operators** <, > or =

```
Student s1 = Student("Michael", "Johnson", 1998, 54321, 4800);

Student s2 = Student("William", "Taylor", 1999, 35791, 6000);
```

54321 > 35791

For example, let's assume our **comparison criterion** is based on the **student number in ascending order**: *s1 is* ***greater than*** *(follows) s2*

Sorting problem and algorithms

We cannot simply **compare** *s1* and *s2* with **relational operators** <, > or =

```
Student s1 = Student("Michael", "Johnson", 1998, 54321, 4800);

Student s2 = Student("William", "Taylor", 1999, 35791, 6000);
```

Johnson precedes Taylor

For example, let's assume our **comparison criterion** is based on the **student surname in ascending order**: *s1 is **less than** (precedes) s2*

Sorting problem and algorithms

We cannot simply **compare** *s1* and *s2* with **relational operators** <, > or =

```
Student s1 = Student("Michael", "Johnson", 1998, 54321, 4800);

Student s2 = Student("William", "Taylor", 1999, 35791, 6000);
```

How can we define those **comparison criteria** in Java?

interface Comparator<T>

```
public interface Comparator<T> {
    …
    int compare(T o1, T o2);
}
```

Generic interface for a type T

o1: the first object to be compared.
o2: the second object to be compared.

Return a **negative integer**, **zero**, or a **positive integer** as the *first argument* is **less than**, **equal to**, or **greater than** the *second*.

interface Comparator<*Student*>

```
public interface Comparator<Student> {
    …
    int compare(Student o1, Student o2);
}
```

Specific interface for a type *Student*

o1: the first object to be compared.
o2: the second object to be compared.

Return a **negative integer**, **zero**, or a **positive integer** as the *first argument* is **less than**, **equal to**, or **greater than** the *second*.

interface Comparator<*Student*>

```
public interface Comparator<Student> {
    …
    int compare(Student o1, Student o2);
}
```

We need to define a **class** that describes the **comparison criterion** for *Students* based on this **interface contract**.

o1: the first object to be compared.
o2: the second object to be compared.

Return a **negative integer**, **zero**, or a **positive integer** as the *first argument* is **less than**, **equal to**, or **greater than** the *second*.

interface Comparator<*Student*>

```
public interface Comparator<Student> {

    …

    int compare(Student o1, Student o2);
}
```

Let's write a **class** that defines this **contract** for comparing *Student* objects according to their *studentNumber* attribute.

o1: the first object to be compared.
o2: the second object to be compared.

Return a **negative integer**, **zero**, or a **positive integer** as the *first argument* is **less than**, **equal to**, or **greater than** the *second*.

interface Comparator<*Student*>

```
public class StudentNumberComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
        // code to write based on the contract



    }
}
```

Return a **negative integer**, **zero**, or a **positive integer** as the *first argument* is **less** than, **equal** to, or **greater** than the *second*.

interface Comparator<*Student*>

```
public class StudentNumberComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
        if (o1.getStudentNumber() < o2.getStudentNumber())
            return -1;




    }
}
```

Return a **negative integer** as the *first argument* is **less than** the *second*.

interface Comparator<Student>

```
public class StudentNumberComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
        if (o1.getStudentNumber() < o2.getStudentNumber())
           return -1;
        else if (o1.getStudentNumber() > o2.getStudentNumber())
           return 1;


    }
}
```

Return a **positive integer** as the *first argument* is **greater than** the *second*.

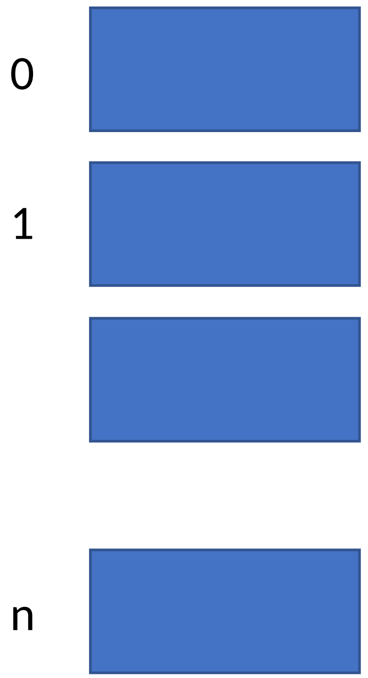
interface Comparator<*Student*>

```
public class StudentNumberComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
        if (o1.getStudentNumber() < o2.getStudentNumber())
            return -1;
        else if (o1.getStudentNumber() > o2.getStudentNumber())
            return 1;
        else
            return 0;
    }
}
```

Return **zero** as the *first argument* is **equal to** the *second*.

Collections.sort: Comparator

- A sorting algorithm **arranges elements** of a list **into an order**.
- Now, we can compare *Student* objects using *StudentNumberComparator*



```
Student("Michael", "Johnson", 1998, 54321, 4800);
Student("William", "Taylor", 1999, 35791, 6000);
Student("Elisabeth", "Smith", 1995, 12345, 5000);
Student("John", "Doe", 2001, 45678, 5600);
```

Collections.sort: Comparator

- Do we need to write **our** sorting algorithm?
- No, the *collection framework* already has a *sort* method we can use

```
{
    List<Student> students = new ArrayList<>();
    Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
    Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
    Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

    students.add(s1);
    students.add(s2);
    students.add(s3);

    Collections.sort(students, new StudentNumberComparator()); // sorting based on studentNumber

}
```

- We just pass to *sort* a **list** and a class that defines the desired **comparison criterion** based on the *Comparator* interface

Collections.sort: Comparator

- Do we need to write **our** sorting algorithm?
- No, the *collection framework* already has a *sort* method we can use

```
{
    List<Student> students = new ArrayList<>();
    Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
    Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
    Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

    students.add(s1);
    students.add(s2);
    students.add(s3);

    Collections.sort(students, new StudentSurnameComparator()); // sorting based on surname

}
```

- We just pass to *sort* a **list** and a class that defines the desired **comparison criterion** based on the *Comparator* interface

Outline

- Java Collections Framework
  - Introduction
  - Lists
  - Maps
- Sorting Collections
  - Comparator interface
  - **Comparable interface**
- Streams and Files
  - Standard Streams and System class
  - Writing to and Reading text data from a File
  - try-with-resources

Natural Comparison

- Some classes of objects can have a *natural comparison criterion*
- By default, String objects are compared in lexicographically ascending order

- This can be implemented for a custom class of objects using another **interface contract**, called *Comparable*, that the class implements
- For example, we can assume *Student* objects should be compared by default using the *surname* attribute

interface Comparable<T>

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Generic interface for a type T

other: the object to be compared.

Compares **this** object with the specified object **other** 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.

interface Comparable<*Student*>

```
public interface Comparable<Student> {
    int compareTo(Student o);
}
```

Specific interface for *Student* type

o: the object to be compared.

Compares **this** object with the specified object **other** 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.

interface Comparable<Student>

```
public class Student extends Person implements Comparable<Student> {
    ...
    @Override
    public int compareTo(Student o) {
        String surname = super.getSurname(); // getting surname for this object
        String otherSurname = o.getSurname();
        return surname.compareTo(otherSurname); // defined in the String class
    }
}
```

- s1.*compareTo*(s2) will return a *negative* number, *positive* number or *zero*
- We used *compareTo* from the *String* class to return the result according to the objects' *surname* attributes

Collections.sort: Natural Comparison

We just pass to *sort* the **list**, and it will use the *natural comparison* criterion defined in the *Student* class (must be defined, otherwise will raise an error)

```
{
    List<Student> students = new ArrayList<>();
    Student s1 = new Student("Michael", "Johnson", 1998, 54321, 4800);
    Student s2 = new Student("William", "Taylor", 1999, 35791, 6000);
    Student s3 = new Student("Elisabeth", "Smith", 1995, 12345, 5000);

    students.add(s1);
    students.add(s2);
    students.add(s3);

    Collections.sort(students) // sorting based on natural comparison (surname)

}
```

Outline

- Java Collections Framework
  - Introduction
  - Lists
  - Maps
- Sorting Collections
  - Comparator interface
  - Comparable interface
- Streams and Files
  - **Standard Streams and System class**
  - Writing to and Reading text data from a File
  - try-with-resources

Streams

Program Data

stored on the **stack** and **heap** as **primitive** and **reference types**

Streams

Program
Data

stored on the **stack** and **heap** as **primitive** and **reference types**

How does a program **communicate** with its environment?

Streams

Program Data

stored on the **stack** and **heap** as **primitive** and **reference types**

⌨

🖥

It **exchanges** data (bytes) with I/O devices, commonly the *keyboard* and the *display*

Streams

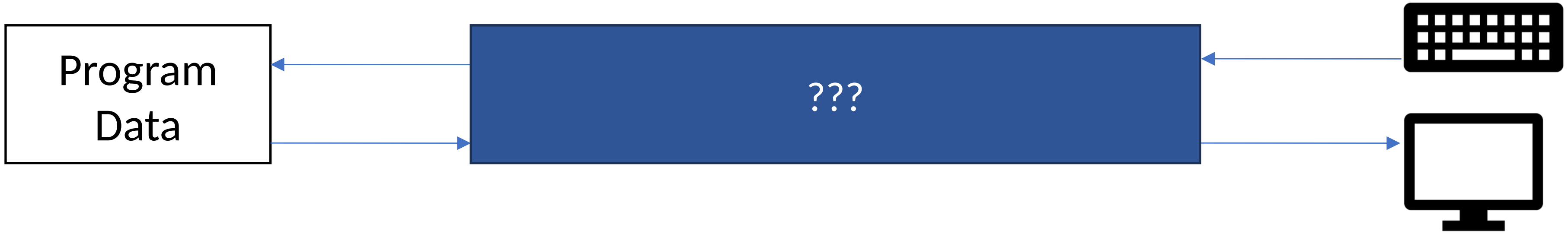
Program Data

stored on the **stack** and **heap** as **primitive** and **reference types**

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

It **exchanges** data (bytes) with I/O devices, commonly the *keyboard* and the *display*

Streams



Program Data

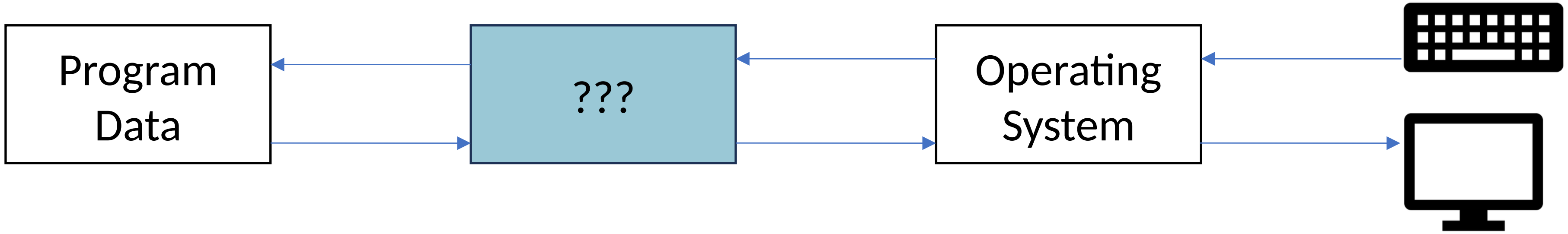
???

stored on the **stack** and **heap** as **primitive** and **reference types**

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

How does the **data exchange** between the *program* and the *keyboard/display* **happen**?

Streams



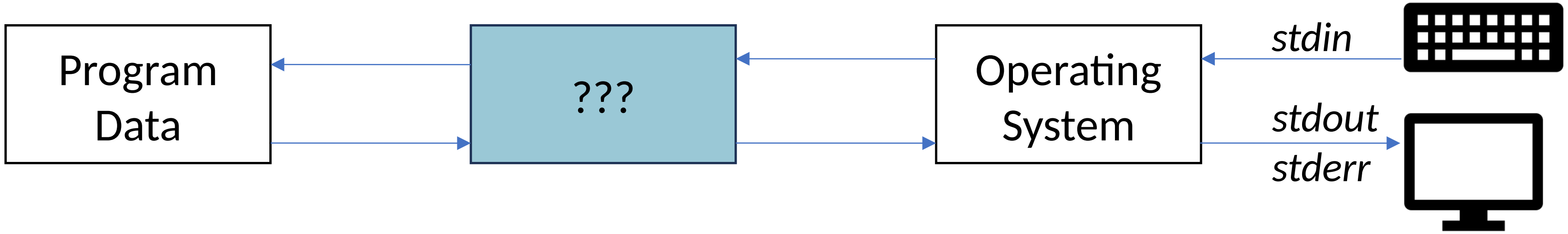
stored on the **stack** and **heap** as **primitive** and **reference types**

streams: abstract access to *environment*

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

A **stream** is an **abstraction** the operating system (OS) provides, representing a **sequence of bytes** exchanged between a *program* and its *environment*, e.g., the **keyboard** and the **display**.

Streams



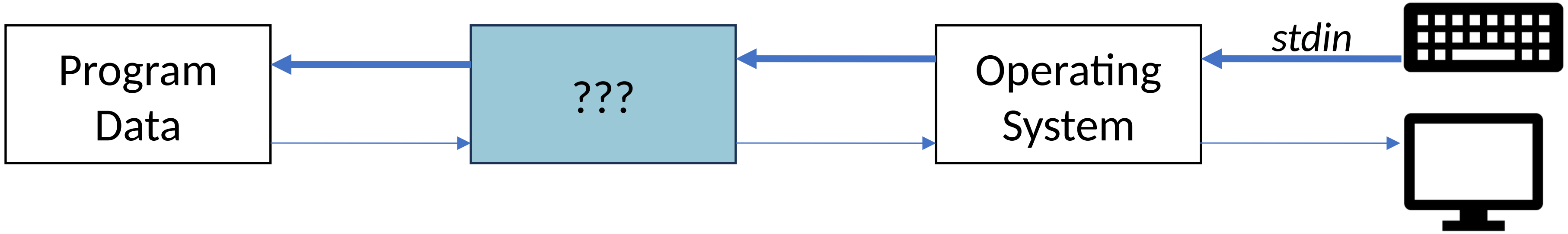
stored on the **stack** and **heap** as **primitive** and **reference types**

streams: *stdin*, *stdout* and *stderr*

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

The OS provides three **standard streams** by default—**standard input** (stdin), **standard output** (stdout), and **standard error** (stderr)—**connected** to the *process* (program)

Streams



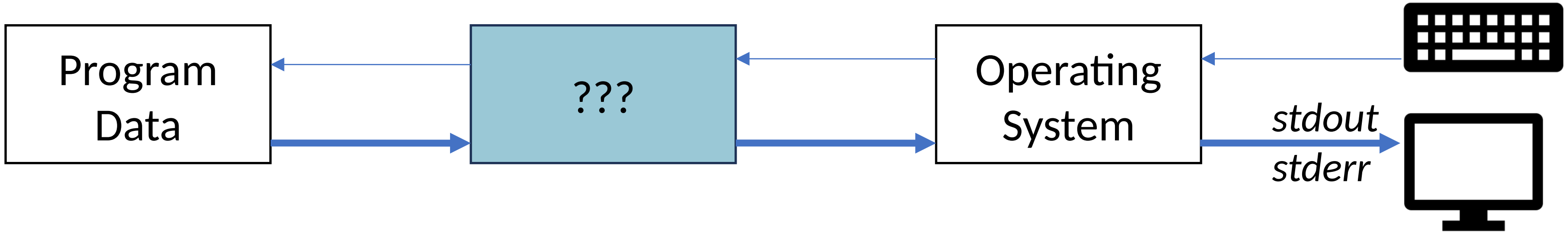
stored on the **stack** and **heap** as **primitive** and **reference types**

streams: *stdin*, *stdout* and *stderr*

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

The **standard input stream (stdin)** allows a program to **read** *input* data from the keyboard abstractly, i.e., without knowing the low-level hardware details

Streams



Program Data

???

Operating System

stdout

stderr

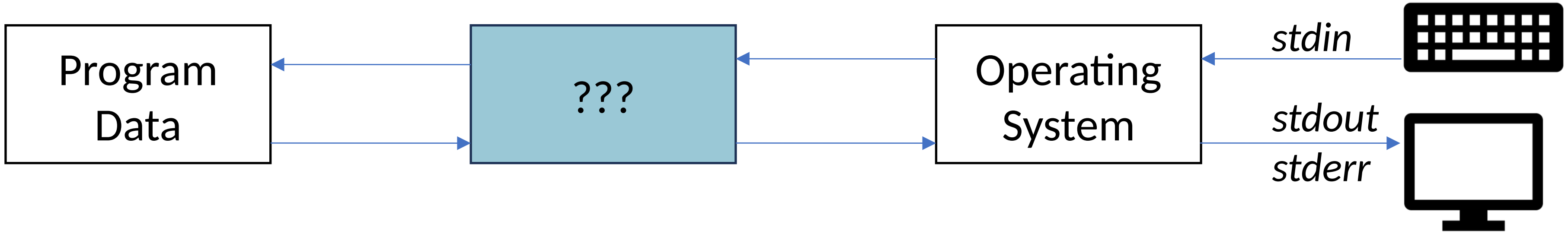
stored on the **stack** and **heap** as **primitive** and **reference types**

streams: *stdin*, *stdout* and *stderr*

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

The **standard out (stdout)** and **error streams (stderr)** allow a program to **write** *output* and *error* data on the display **abstractly**, i.e., without knowing the low-level hardware details

Streams



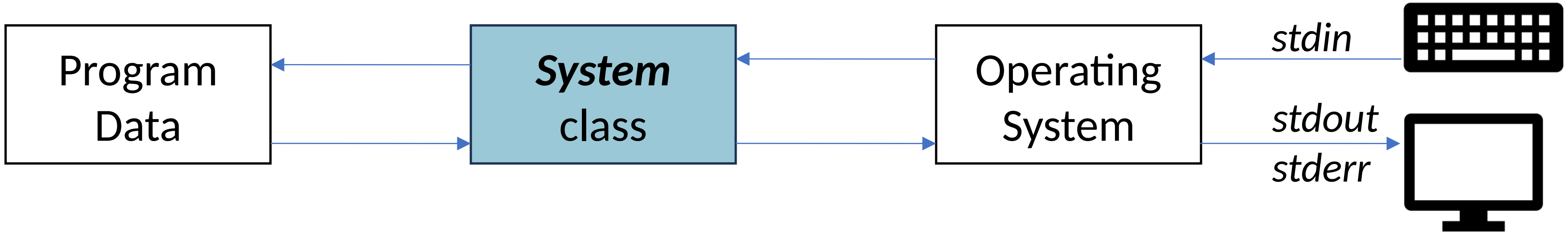
stored on the **stack** and **heap** as **primitive** and **reference types**

streams: *stdin*, *stdout* and *stderr*

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

How do we use those OS **standard streams** from a **Java program**?

Streams



Program Data

System class

Operating System

stdin

stdout

stderr

stored on the **stack** and **heap** as **primitive** and **reference types**

provides access to *the OS streams*

streams: *stdin*, *stdout* and *stderr*

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

The Java *System* class allows **access** to the OS' **standard streams**

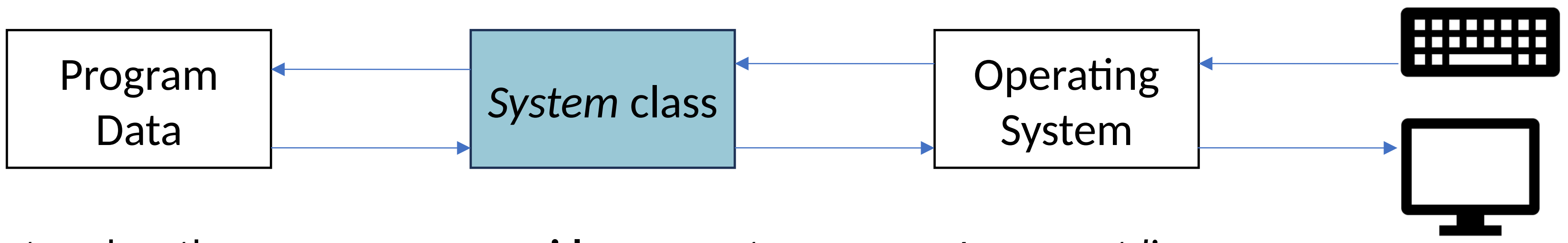
Streams: *System* class

The *System* class provides an *abstract* way to interact with I/O devices via **standard streams**:

- *System.in*
- *System.out*
- *System.err*

Streams: *System* class

- When using *System.in*, *System.out and System.err* data is **appropriately converted** to/from the OS' *default character encoding*
- This **ensures** that the data **read from** or **written to** these streams is **correctly interpreted**



stored on the **stack** and **heap** as **primitive** and **reference types**

provides access to *the OS streams*

streams: *stdin*, *stdout* and *stderr*

OS programs like the *console* and *text editors* use a **standard character encoding**, e.g., *UTF-8*

Streams: *System* class (out)

- *System*.*out* is a *public static PrintStream* object in the *System* class
- It allows data **to be sent** to the *console* via the *standard output* (stdout)

```
public final class System {
    private System() {
    }

    public static final InputStream in;
    public static final PrintStream out;
    public static final PrintStream err;
    ...
}
```

Streams: *System.out.println*

- When we write the instruction:

  System.out.println("Hello, World!");

- We **call** the *println* method on the *PrintStream* instance referred to by *System.out*
- The method **encodes** the *String* object "Hello, World!" into **bytes** using the platform's default *character encoding* (e.g., UTF-8)
- These bytes are then **sent** to the *standard output*, which **displays** a message on the *console*.

Streams: *System* class (in)

- *System.in* is a *public static InputStream* object in the *System* class
- It allows **input data to be read** from the *console* via *standard input* (stdin)

```
public final class System {
    private System() {
    }

    public static final InputStream in;
    public static final PrintStream out;
    public static final PrintStream err;
    …
}
```

Streams: *Scanner(System.in)*

- When we write the instruction:

```
Scanner scanner = new Scanner(System.in);
```

- *System.in* **reads** *raw bytes* from the *standard input* (keyboard)
- The *scanner* object **decodes** these bytes into **characters** using the platform's *default encoding* (e.g., UTF-8)
- The *scanner* object **parses** this **text data** and reads it as **different types** (e.g., *nextLine*(), *nextInt*(), *nextDouble*(), etc.)

Outline

- Java Collections Framework
  - Introduction
  - Lists
  - Maps
- Sorting Collections
  - Comparator interface
  - Comparable interface
- Streams and Files
  - Standard Streams and System class
  - **Writing to and Reading text data from a File**
  - try-with-resources

Streams: Other Abstractions

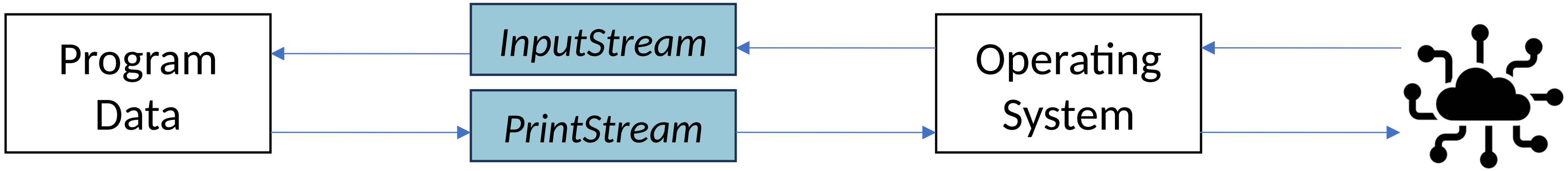
The *System.in (InputStream)*, *System.out* and *System.err* (*PrintStream*) objects allow communication with the *keyboard* and the *screen*.

Can we:

- Use a *PrintStream* object to **send** any data type to a destination other than the *standard output* or *standard error* (screen)?
- Use an *InputStream* object to **read** any data from a source other than the *standard input* (keyboard)?

Yes, similar **stream objects** can be created. These streams provide an **abstraction** that allows **writing to** and **reading from** various **sources** and **destinations**, such as *network connections*, *files*, and other *storage devices*.

Streams: Network Abstraction



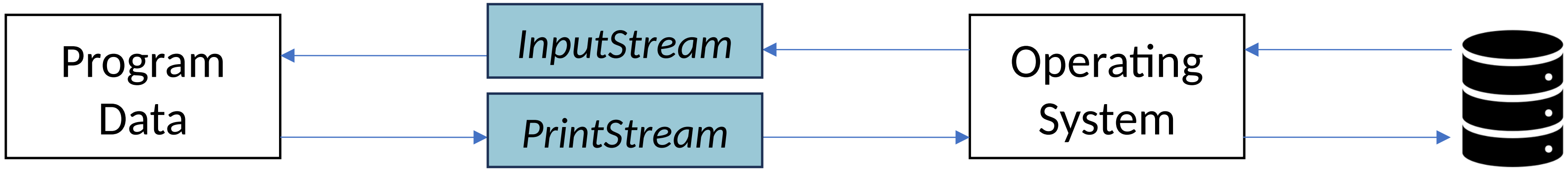
stored on the **stack** and **heap**

abstracts access to *Network*

accesses *Network*

A **stream** is an **abstraction** the operating system (OS) provides, representing a **sequence of bytes** exchanged between a *program* and its *environment*, e.g., the **network**.

Streams: Storage Abstraction



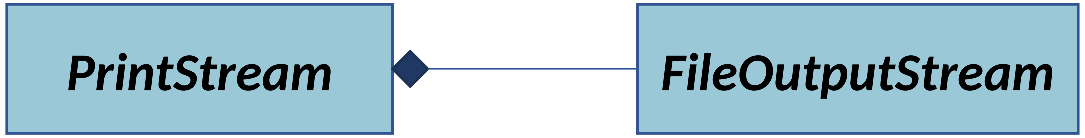
stored on the **stack** and **heap**

abstracts access to *Storage*

accesses *Storage*

A **stream** is an **abstraction** the operating system (OS) provides, representing a **sequence of bytes** exchanged between a *program* and its *environment*, e.g., the **storage**.

Streams: Write Text to a File



We could create a *PrintStream* object that **writes to a file** instead of the *stdout* (or *stderr*) via a composite *FileOutputStream* object:

```
PrintStream out = new PrintStream(new FileOutputStream("example.txt"));
```

Optimised to deal
with *text data*

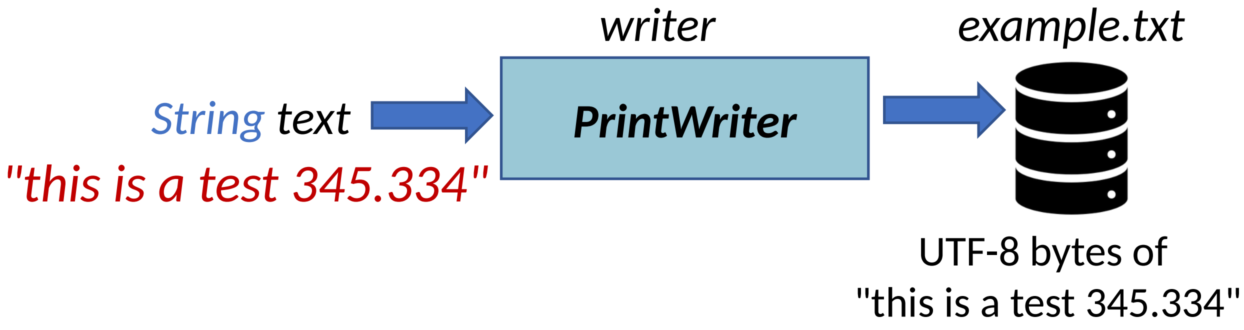
PrintWriter is preferable when dealing with **text** data:

```
PrintWriter writer = new PrintWriter("example.txt"));
```

Streams: *PrintWriter* example

```
{
 PrintWriter writer = null;
 String fileName = "example.txt";
 try {
     writer = new PrintWriter(fileName);
     String text = "this is a test " + 345.334;
     writer.println(text); // writes to the file instead of a standard stream
 } catch (IOException e) {
     System.out.println("Error while accessing the file" + e.getMessage());
  } finally {
     if (writer != null) {
        writer.close(); // flushes the buffer and releases the resources
     }
   }
 System.out.println("End of the program");
 }
```

This can generate a **checked exception**: the compiler **forces** us to use try-catch



writer

example.txt

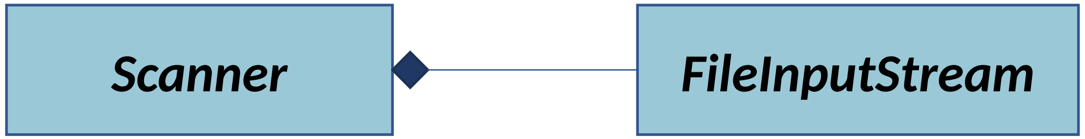
String text

"this is a test 345.334"

PrintWriter

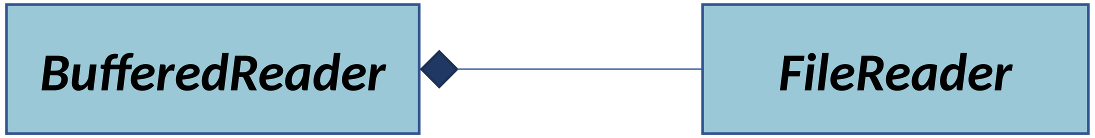
UTF-8 bytes of
"this is a test 345.334"

Streams: Read Text from a File



We could create a *Scanner* object that **reads from a file** instead of the *stdin* via a composite *FileInputStream* (a subclass of *InputStream*) object:

```
Scanner scanner = new Scanner(new FileInputStream("example.txt"));
```



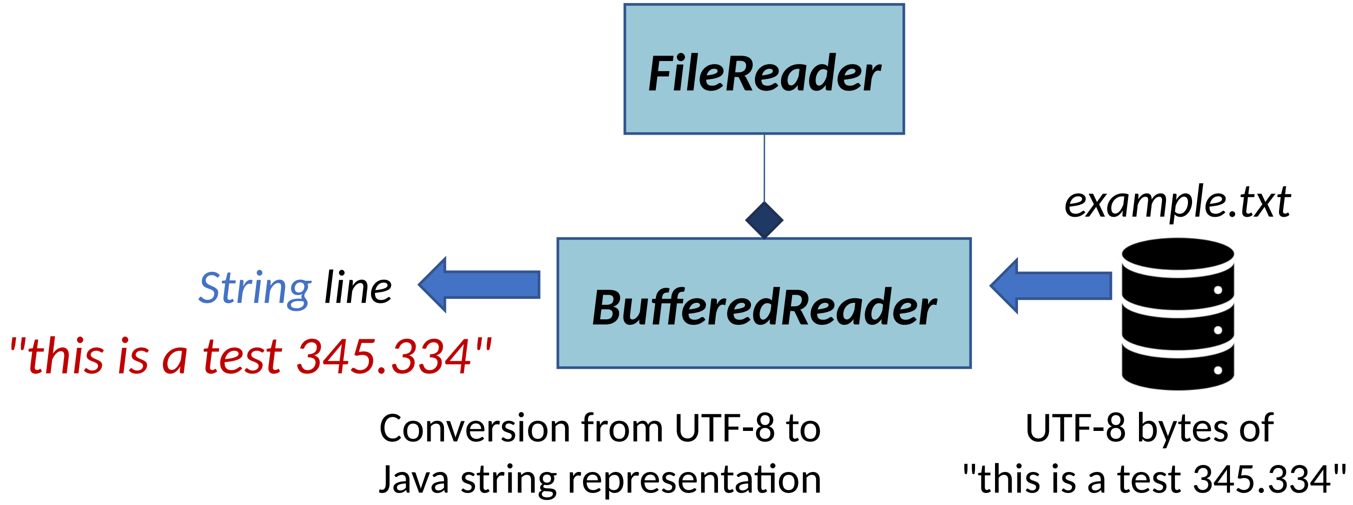
Optimised to deal
with *text data*

FileReader combined with *BufferedReader* is preferable when dealing with **text** data:

```
BufferedReader reader = new BufferedReader(new FileReader("example.txt"));
```

Streams: *BufferedReader* example

```
{
   BufferedReader reader = null;
   try {
       reader = new BufferedReader(new FileReader("example.txt"));
       String line;
       while ((line = reader.readLine()) != null) {
           System.out.println(line);
       }
   } catch (IOException e) {
        System.out.println("Error while accessing the file" + e.getMessage());
      } finally {
         if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
               System.out.println("Error");
              }
          }
       }
}
```



Conversion from UTF-8 to Java string representation

UTF-8 bytes of "this is a test 345.334"

Outline

- Java Collections Framework
  - Introduction
  - Lists
  - Maps
- Sorting Collections
  - Comparator interface
  - Comparable interface
- Streams and Files
  - Standard Streams and System class
  - Writing to and Reading text data from a File
  - **try-with-resources**

Exceptions: try-with-resources

```
{
  PrintWriter writer = null;
  String fileName = "example.txt";
  try {
      writer = new PrintWriter(fileName);
      String text = "this is a test " + 345.334;
      writer.println(text); // writes to the file instead of a standard stream
  } catch (IOException e) {
      System.out.println("Error while accessing the file" + e.getMessage());
   } finally {
      if (writer != null) {
         writer.close(); // flushes the buffer and releases the resources
      }
    }
  System.out.println("End of the program");
  }
```

We need to **release resources explicitly** calling *writer.close()* in the *finally* block

Exceptions: try-with-resources

```
{
 String fileName = "example.txt";
 try (PrintWriter writer = new PrintWriter(fileName)) {
     String text = "this is a test " + 345.334;
     writer.println(text); // writes to the file instead of a standard stream
 } catch (IOException e) {
     System.out.println("Error while accessing the file" + e.getMessage());
  }

 System.out.println("End of the program");
 }
```

Resource declared within the parentheses are **automatically closed** when the *try* block **exits** (**normally** or due to an **exception**)

No need for an explicit *finally* block to close resources