SWEN20003

Workshop 8, Week 9

Eleanor McMurtry, University of Melbourne

Part 1: Generic data structures

Avoiding arrays for fun and profit

- Java arrays are not flexible:

Avoiding arrays for fun and profit

- Java arrays are not flexible:
  - fixed size
  - one kind of structure
  - homogeneous

Avoiding arrays for fun and profit

- Java arrays are not flexible:
  - fixed size
  - one kind of structure
  - homogeneous
- In C, we'd have to define other structures (like a linked list) ourselves.

Java generic library

- Java provides a library of generic data structures we can use.

Java generic library

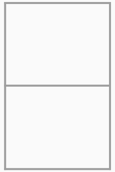
- Java provides a library of generic data structures we can use.
- We focus on two:
  - `ArrayList`: auto-resizing array
  - `HashMap`: dictionary

ArrayList

```
List<Double> list = new ArrayList<>();
```

interface implemented by `ArrayList`

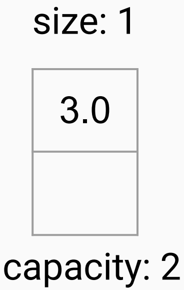
size: 0



capacity: 2

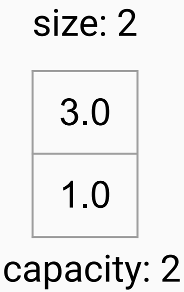
ArrayList

```
List<Double> list = new ArrayList<>();

list.add(3.0);
```



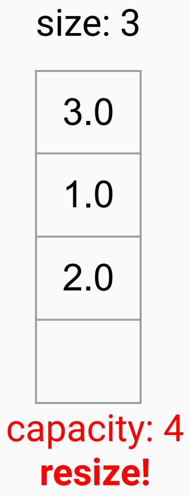
ArrayList

```
List<Double> list = new ArrayList<>();

list.add(3.0);

list.add(1.0);
```



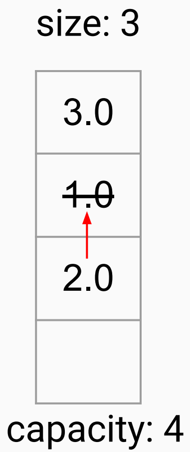
ArrayList

```
List<Double> list = new ArrayList<>();

list.add(3.0);

list.add(1.0);

list.add(2.0);
```



ArrayList

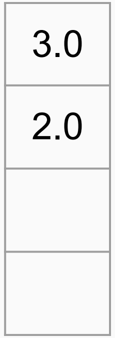
```
List<Double> list = new ArrayList<>();

list.add(3.0);

list.add(1.0);

list.add(2.0);

list.remove(1);
```



ArrayList

```java
List<Double> list = new ArrayList<>();
list.add(3.0);
list.add(1.0);
list.add(2.0);
list.remove(1);
```

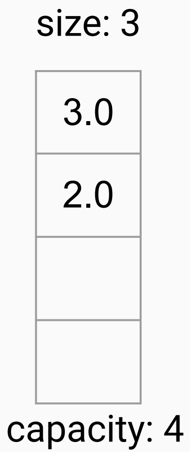
size: 3



capacity: 4

ArrayList

```
List<Double> list = new ArrayList<>();

list.add(3.0);

list.add(1.0);

list.add(2.0);

list.remove(1);

list.get(1); // = 2.0
```



Using `ArrayList`

- `ArrayLists` are useful when there is no maximum number of elements
- Will typically be your go-to data structure

HashMap

```
Map<String, Integer> phonebook = new HashMap<>();
```

Name	Number

HashMap

```
Map<String, String> phonebook = new HashMap<>();

phonebook.put(“Alice”, “95534443”);
```

“Alice”.hashCode() = 3

Name	Number
Alice	95534443

HashMap

```
Map<String, String> phonebook = new HashMap<>();

phonebook.put(“Alice”, “95534443”);

phonebook.put(“Bob”, “93244221”);
```

“Bob”.hashCode() = 1

Name	**Number**
Bob	93244221
Alice	95534443

HashMap

```
Map<String, String> phonebook = new HashMap<>();

phonebook.put("Alice", "95534443");

phonebook.put("Bob", "93244221");

phonebook.get("Alice"); // = "95534443"
```

Name	**Number**
Bob	93244221
Alice	95534443

HashMap

```
Map<String, String> phonebook = new HashMap<>();

phonebook.put(“Alice”, “95534443”);

phonebook.put(“Bob”, “93244221”);

phonebook.get(“Alice”); // = “95534443”

phonebook.get(“Charlie”); // = null
```

Name	Number
Bob	93244221
Alice	95534443

Using HashMap

- HashMaps are useful when you need to look up objects by a key e.g. `Student`s by `name`.

```
Map<String, Student> students = new HashMap<>();
Student alice = new Student("Alice", "759332");
students.put(alice.getName(), alice);
```

Using HashMap

- HashMaps are useful when you need to look up objects by a key e.g. `Student`s by `name`.

```
Map<String, Student> students = new HashMap<>();
Student alice = new Student("Alice", "759332");
students.put(alice.getName(), alice);
```

- Classes used as the **value** of a `HashMap` need to override `hashCode()` and `equals()`.

Using HashMap

- Classes used as the **value** of a `HashMap` need to override `hashCode()` and `equals()`.

```
public class Student {
    public int hashCode() {
        return (name + id).hashCode();
    }
    public boolean equals(Object rhs) {
        if (rhs instanceof Student) {
            return ((Student)rhs).id == id &&
                   ((Student)rhs).name.equals(name);
        } else {
            return false;
        }
    }
}
```

Part 2: Design patterns

Design pattern philosophy

- How many times have you thought “surely somebody has solved this problem before?”

Design pattern philosophy

- How many times have you thought “surely somebody has solved this problem before?”
- Design patterns give a structured solution to common problems.

Example 1: **Singleton**

- The **Singleton** pattern is used for a class that contains **global state** and is only instantiated once.

```java
public class ServerConnection {
    private ServerConnection() { }
    private static ServerConnection _INSTANCE;
    public static ServerConnection getInstance() {
        if (_INSTANCE == null) {
            _INSTANCE = new ServerConnection();
        }

        return _INSTANCE;
    }
}
```

Example 1: **Singleton**

- The **Singleton** pattern is used for a class that contains **global state** and is only instantiated once.
- As with any global state, overuse of Singletons is a bad idea.

```java
public class ServerConnection {
    private ServerConnection() { }
    private static ServerConnection _INSTANCE;
    public static ServerConnection getInstance() {
        if (_INSTANCE == null) {
            _INSTANCE = new ServerConnection();
        }

        return _INSTANCE;
    }
}
```

Example 2: **Template**

- A base class provides outline of workflow, and derived classes implement specific methods.

```java
abstract class ProtocolInteraction {
    public void execute() {
        String request = this.generateRequest();
        // send request here
        String data = this.processResponse("response");
        this.saveToDatabase(data);
    }

    public abstract String generateRequest();
    public abstract String processResponse(String response);
    public abstract void saveToDatabase(String data);
}
```

Example 2: **Template**

```java
class HttpInteraction extends ProtocolInteraction {
    @Override
    public String generateRequest() {
        return "GET /api/data HTTP/1.1";
    }

    @Override
    public String processResponse(String response) {
        if (response.contains("200 OK")) {
            return "important data";
        } else {
            return "error";
        }
    }

    @Override
    public void saveToDatabase(String data) {
        System.out.println(data);
    }
}
```

```java
class SqlInteraction extends ProtocolInteraction {
    @Override
    public String generateRequest() {
        return "SELECT StudentName FROM students;";
    }

    @Override
    public String processResponse(String response) {
        return response;
    }

    @Override
    public void saveToDatabase(String data) {
        for (String name : data.split(",")) {
            System.out.println(name);
        }
    }
}
```

Example 2: **Template**

- Templates separate out the specific parts of an algorithm or procedure.

- This makes varying implementations easier to read.

Example 3: **Strategy**

- A class performs some task, but delegates the implementation to an interface.

```
interface IPricingStrategy {
    double discount(double price);
}

class Bar {
    public double calculatePrice(String drink, IPricingStrategy pricingStrategy) {
        switch (drink) {
            case "water":
                return 0;
            case "soft drink":
                return pricingStrategy.discount(6.00);
            case "ginger beer":
                return pricingStrategy.discount(9.50);
            default:
                return 0;
        }
    }
}
```

Strategy vs Template

- Not using inheritance means the strategy can be changed **at runtime**
- A **weaker relationship** => weaker coupling => more general design

Part 3: Exceptions

Types of errors

- **Syntax error:** code is not valid Java

Types of errors

- **Syntax error:** code is not valid Java
- **Semantic error:** code compiles, but does not do what was intended

Types of errors

- **Syntax error:** code is not valid Java
- **Semantic error:** code compiles, but does not do what was intended
- **Runtime error:** program detects invalid state and exits early (e.g. `NullPointerException`)

Handling runtime errors

- **OS-level:** kernel signals that crash program: SIGSEGV (segmentation fault), SIGFPE (floating point error e.g. divide-by-zero), …

Handling runtime errors

- **OS-level:** kernel signals that crash program: SIGSEGV (segmentation fault), SIGFPE (floating point error e.g. divide-by-zero), …
- **C-style:** check return codes of functions

Handling runtime errors

- **OS-level:** kernel signals that crash program: SIGSEGV (segmentation fault), SIGFPE (floating point error e.g. divide-by-zero), …
- **C-style:** check return codes of functions
- **Defensive programming:** test for errors before attempting action

Exceptions!

- **Exceptions:** create an object representing the error, and unwind the call stack until the error is handled

Exceptions!

Call Stack

```
main()
```

Exceptions!

Call Stack

```
main()

handleCustomer()
```

Exceptions!

Call Stack

```
main()

handleCustomer()

buyDrink()
```

Exceptions!

Call Stack

```
main()

handleCustomer()

buyDrink()

calculatePrice()
```

Exceptions!

Call Stack

```
main()

handleCustomer()

buyDrink()

calculatePrice()
```

```
throw new InvalidArgumentException("unknown drink " + drink);
```

Exceptions!

Call Stack

```
main()

handleCustomer()

buyDrink()
```

~~calculatePrice()~~

```
throw new InvalidArgumentException("unknown drink " + drink);
```

Exceptions!

Call Stack

```
main()

handleCustomer()
```

~~buyDrink()~~

~~calculatePrice()~~

```
throw new InvalidArgumentException("unknown drink " + drink);
```

Exceptions!

Call Stack

```
main()

handleCustomer()
```

~~buyDrink()~~

~~calculatePrice()~~

```
try {
    buyDrink(drink);
} catch (InvalidArgumentException e) {
    System.out.println(e.getMessage());
}
```

```
throw new InvalidArgumentException(“unknown
drink “ + drink);
```

Why exceptions?

- Handles an unexpected or “exceptional” state

Why exceptions?

- Handles an unexpected or “exceptional” state
- Lets the user of the code choose how to handle the error

Why exceptions?

- Handles an unexpected or “exceptional” state
- Lets the user of the code choose how to handle the error
- Do not need to remember to check return codes, or defensively program

Why exceptions?

- Handles an unexpected or “exceptional” state
- Lets the user of the code choose how to handle the error
- Do not need to remember to check return codes, or defensively program
- Make it clear where errors can and cannot happen

When to use exceptions

- Exceptions should be used when **a method cannot recover** from an unusual state, but the caller might be able to.

When to use exceptions

- Exceptions should be used when **a method cannot recover** from an unusual state, but the caller might be able to.
- An unrecoverable error should not be an exception.

Demonstration