Table of Contents

  - Subclass
  - Gist
  - Example: Chess
  - Super Keyword
  - Shadowing
  - Overriding
    - Why?
- L10 - Polymorphism and Abstract Classes
  - The Four Principles of Object Oriented Programming
    - Encapsulation
    - Data Abstraction
    - Polymorphism
    - Inheritance
  - The Object Class
  - `instanceof` Keyword
  - Upcasting
  - Downcasting
  - Polymorphism
  - Abstract Methods
    - Definition
  - Abstract Classes
  - Concrete Classes
- L11 - Interfaces
  - How to Use Interfaces
  - Default Methods
  - Extending Interfaces
  - A Common Use - Sorting - `Comparable<T>`
- L12 - Unified Modelling Language (UML)
  - What is it?
  - Class Representation
    - Class Attributes
    - Class Methods
  - Class Relationships
    - Association
    - Multiplicity
    - Example
    - Aggregation
    - Composition
  - Generalization - Inheritance
- L13 - Generics
  - Introduction
  - The Comparable Interface
  - What does T mean?
  - ArrayList
  - Using the ArrayList Class
- L14 - Exceptions

  - Types of Errors
  - Runtime Errors
    - Best Solution
  - Exceptions
    - Exception Handling
    - Generating Exceptions
    - Defining Exceptions
    - Chaining Exceptions
    - Types of Exceptions
- L17 - Software Testing and Design
  - Unit Testing
    - Why unit testing?
- L18/L19 - Design Patterns
  - Analysing and Publishing a Pattern
  - Common Design Patterns
    - Template Method
    - Strategy Pattern
    - Factory Method Pattern
    - Observer Pattern
- L20 - Advanced Java and OOP Concepts
  - Enums (enumerated types)
    - Usage
    - Accessing
    - Tying other Information
  - Variadic Parameters
  - Functional Interfaces
  - Lambda Expressions
  - Method Reference
  - Streams
    - Operations
  - You will not be expected to write code on anything from these notes on this lecture.
- L21 - Games
  - Sequential Programming
  - Event Driven Programming
    - Examples of event-driven systems
  - Asynchronous Programming
  - Entity Component Approach
- Exam Tips
  - Week 1
  - Week 2
  - Week 3
  - Week 4
  - Week 5
  - Week 6
  - Week 7

Operations

The addition operator `+` can concatenate the string representation of two *objects* (using the `toString()` method inherited by every class)

```
int a = 1;
String s = "a = " + a; /* good */
```

Keep in mind:

```
"1 + 1 = " + 1 + 1 /* equates to 1 + 1 = 11 */
"1 + 1 = " + (1 + 1) /* equates to 1 + 1 = 2 */
```

Operator precedence still holds

Immutability

Strings are immutable; once created, they can't be modified, only replaced. An important point to note here is that, while the `String` object is immutable, **its reference variable is not.** So that's why, in the above example, the reference was made to refer to a newly formed `String` object.

```
String str = "knowledge";
String s = str; // assigns a new reference to the same string "knowledge"
str = str.concat(" base"); /* NEW string, prev one is lost cause no other references*/
```

As applications grow, *it's very common for String literals to occupy large area of memory, which can even cause redundancy.* So, in order to make Java more efficient, **the JVM sets aside a special area of memory called the "String constant pool".**

When the compiler sees a `String` literal, it looks for the `String` in the pool. If a match is found, the reference to the new literal is directed to the existing `String` and no new `String` object is created. The existing `String` simply has one more reference. Here comes the point of making `String`objects immutable:

In the `String` constant pool, a `String` object is likely to have one or many references. *If several references point to same String without even knowing it, it would be bad if one of the references modified that String value. That's why String objects are immutable.*

Source and further reading

The String class is marked **final** so that nobody can override the behaviour of its methods.

```java
Integer.parseInt("10");
Double.parseDouble("3.141592");
Boolean.parseDouble("TruE");
```

Boxing/Unboxing

Boxing

The process of converting a primitive to its equivalent wrapper class

Unboxing

The process of converting an instance of a wrapper class to its equivalent primitive type.

Just remember that primitives types are the OG and you're effectively '**boxing**' it up with OOP classes.

Formatting

```java
System.out.format("%2$d %<05d %1$d %3$10s", 10, 22, "Hello");
/*"22 0002210 Hello"
```

L03 - Input and Output (I/O)

← Return to Index

Association

Represents a **has a** (containment) relationship between objects.

When classes are contained by another, we always use an **association**

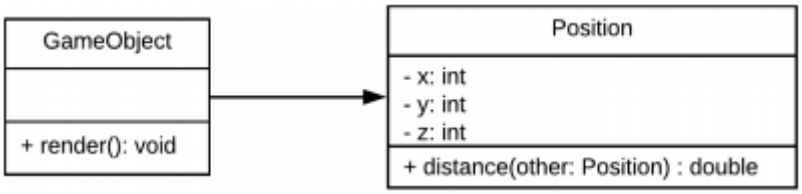

Figure 5: UML Associations

A link indicating **one class contains an attribute that is itself a class**. Does not mean one class "uses" another (in a method, or otherwise).

Multiplicity

Multiplicity on an association specifies the **number of links** that can exist between **instances (objects)** of the associated classes.

Example

Let's create an example for the following scenario:

- A `Student` can take up to five Courses
- A `Student` has to be enrolled in at least one Course
- A `Course` can contain up to 400 Students
- A `Course` should have at least 10 Students

Create a UML representation for the following scenario:

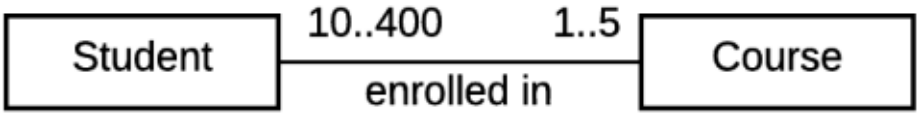

Figure 6: UML Example

- Self association example:
  - Each Student also has a student representative they can contact, who is also a student

```
        array[index+1] = temp;
    }
}
```

This can be further abstracted, to allow bubble sorting with different implementations of `swap` and `outOfOrder`

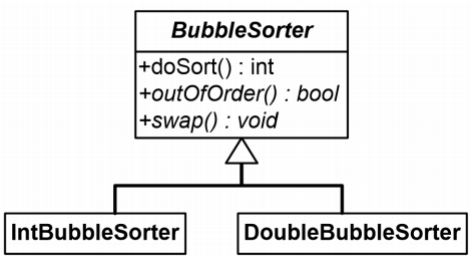

Figure 10: UML Representation of the Template Method Pattern

Here is the template for the abstract bubble sorting class:

```
public abstract class AbstractBubbleSorter {
    private static int operations = 0;
    protected int length = 0;
    protected int doSort(){
        operations = 0;
        if (length <= 1)
            return operations;
        for (int i = length - 2; i >=0; i--){
            for (int j = 0; j <= i; j++){
                if (outOfOrder(j))
                    swap(j);
                operations++;
            }
        }
        return operations;
    }
    protected abstract void swap(int index);
    protected abstract boolean outOfOrder(int index);
}
```

Here is how the `IntBubbleSorter` would work:

```
public class IntBubbleSorter extends AbstractBubbleSorter {
    private int[] array = null;
    public int sort(int[] a){
        array = a;
        length = array.length;
        return doSort();
```