COMP2511 – Course Notes

Contents

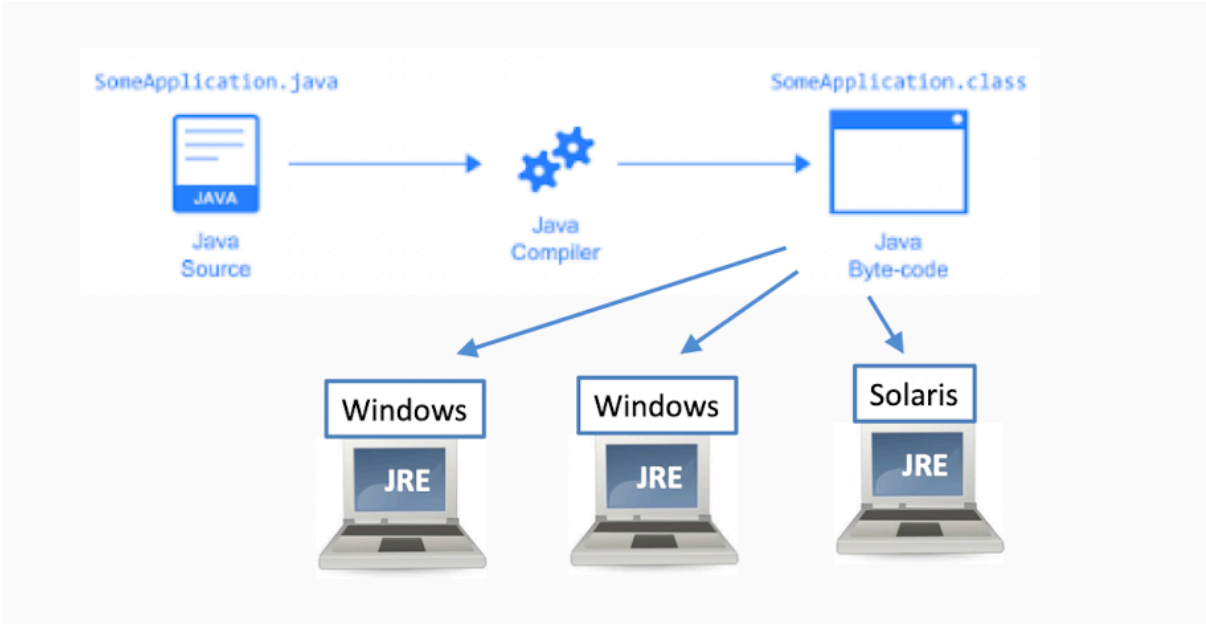
Week 1

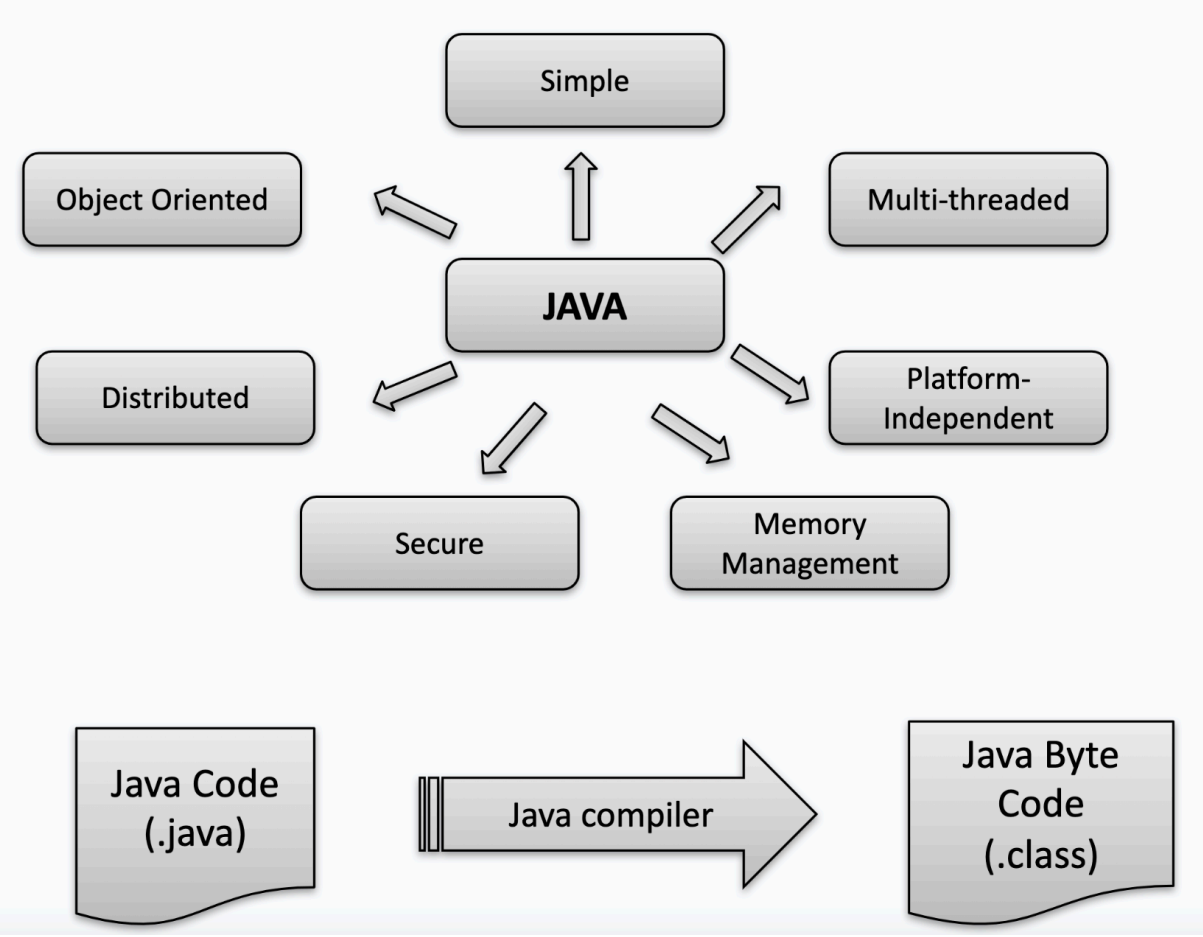
1.1: Course Intro

- The Art of Software Design (OO Design & Programming)

1.2: Introduction to Java

- Java is platform independent





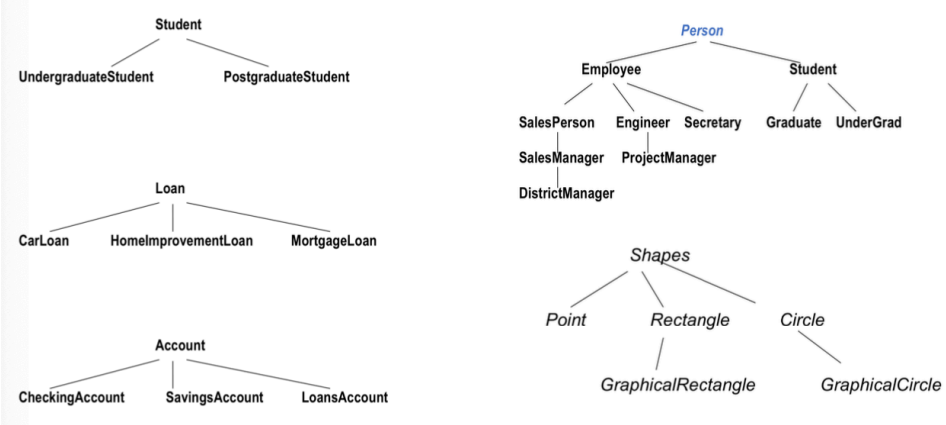
1.3: The Object-Oriented Programming (OOP) in Java
2.1: Inheritance
2.2: Interfaces
2.3: Polymorphism and More OO

Object Oriented Programming (OOP)

- In a procedural programming languages (like 'C'), programming tends to be action-oriented, whereas in Java – programming is objected-oriented.
- In procedural programming:
    - Groups of actions that perform some task are formed into functions and functions are group to form programs.
- In OOP:
    - Programmers create their own user-define types called classes.
    - Each class contains data as well as methods that manipulate the data.
    - An instance of a user-defined type (class) is an object.
    - OOP encapsulates data (attributes) and methods (behaviours) into objects, the data and methods of an object are intimately tied together.
    - Objects have the property of information hiding.

Inheritance in OOP

- Inheritance is a form of software reusability where new classes are created from exisiting classes by absorbing their attributes and behaviours.
- Instead of defining a separate new class, programmers can make it so that a new class inherits the attributes and behaviours of an existing class (called superclass). The new class is a subclass.
- Programmers can add more attributes and behaviours to the subclass.



Is-a Inheritance Relationship

- An object of a subclass can be treated as an object of the superclass
- E.g. UndergraduateStudent is also a Student.
- Use inheritance to model a 'is-a' relationship.
- DO NOT use inheritance unless all or most inherited attributes and methods make sense.

- For example, mathematically a circle ‘is-a’ oval, but should not inherit a class circle from an oval since a oval can change widths and height.

Has-a Association Relationship

- A class object has an object of another class to store its state or do its word.
- It ‘has-a’ reference to that object.
- For example, a Rectangle is-NOT-a Line. But a Rectangle has-a Line.
- ‘Has-a’ relationships are examples of creating new classes by composition of existing classes, as opposed to extending classes, as before.

Designing a class

- Try keeping variables private (local).
- Consider different ways an object can be created.
- Always initialise data.
- If the object is no longer in use, free up all the associate resources.
- Break up class with too many responsibilities, ‘refactoring’.

The class Circle

```
public class Circle  {

    protected static final double pi = 3.14159;
    protected  int  x, y;
    protected  int  r;

   // Very simple constructor
    public Circle(){
        this.x = 1;
        this.y = 1;
        this.r = 1;
    }
    // Another simple constructor
    public Circle(int x, int y, int r){
        this.x = x;
        this.y = y;
        this.r = r;
    }

    /**
     * Below, methods that return the circumference
     * area of the circle
     */
    public  double  circumference( ) {
        return 2 * pi * r ;
    }
    public  double  area ( ) {
        return  pi * r * r ;
    }

}
```

Has-a Approach

- We want to implement GraphicalCircle.
- GraphicalCircle has a Circle.
- It uses methods from the class Circle (area and circumference) to define some of the new methods.
- This is method-forwarding.

```
public class GraphicalCircle2 {
    // here's the math circle
    Circle c;
    // The new graphics variables go here
    Color outline, fill;

    // Very simple constructor
    public GraphicalCircle2() {
        c = new Circle();
        this.outline = Color.black;
        this.fill = Color.white;
    }

    // Another simple constructor
    public GraphicalCircle2(int x, int y, int r,
                            Color o, Color f) {
        c = new Circle(x, y, r);
        this.outline = o;
        this.fill = f;
    }

    // draw method , using object 'c'
    public void draw(Graphics g) {
        g.setColor(outline);
        g.drawOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);
        g.setColor(fill);
        g.fillOval(c.x - c.r, c.y - c.r, 2 * c.r, 2 * c.r);
    }
```

Is-a Approach

- We can say GraphicalCircle is-a Circle.
- Hence, we can define GraphicalCircle as an extension, or subclass of Circle.
- The subclass GraphicalCircle inherits all the variables and methods of its superclass Circle.

```
import java.awt.Color;
import java.awt.Graphics;

public class GraphicalCircle extends Circle {

    Color  outline, fill;
    public GraphicalCircle(){
            super();
            this.outline = Color.black;
            this.fill = Color.white;
    }
    // Another simple constructor
    public GraphicalCircle(int x, int y,
                    int r, Color o, Color f){
        super(x, y, r);
        this.outline = o; this.fill = f;
    }

    public void draw(Graphics  g) {
        g.setColor(outline);
        g.drawOval(x-r, y-r, 2*r, 2*r);
        g.setColor(fill);
        g.fillOval(x-r, y-r, 2*r, 2*r);
    }

}
```

Example

- Note that:

```
GraphicCircle  gc = new GraphicCircle();
      ...
double  area = gc.area();
      ...
Circle  c = gc;
// we cannot call draw method for  "c".
```

Super classes, Objects and the Class Heirarchy

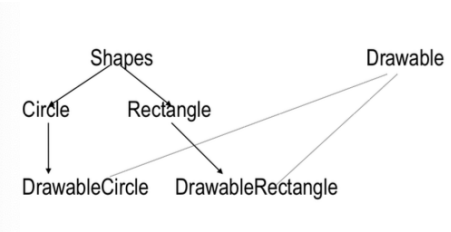
- Every class has a superclass.
- If we don't define a superclass, by default the superclass is the class Object.
- Object is the only class that does not have a superclass.
- Often we need to override the following methods:
    - toString()
    - equals()
    - hasCode(0

Abstract Classes

- Using abstract classes:
    - We can declare classes that define only part of an implementation
    - Leaving extended classes to provide specific implementation of some or all the methods
- Rules:
    - An abstract class is a class declared abstract
    - If a class includes abstract methods, then the class must be declared abstract
    - Abstract classes cannot be instantiated
    - A subclass of an abstract class can be instantiated if it overrides each of the abstract methods of its superclass (abstract) and provides an implementation for all of them
    - If a subclass does not implement all of the abstract methods it inherits, then the subclass is abstract
- In Java, a new class can extend only one superclass

Interfaces

- Interfaces are like abstract classes
- All methods are implicitly abstract (don't need to use abstract keyword)
- Variables are all static and final (constants)
- Just like a class extends its superclass, it can also implement an interface
- In order to implement an interface, the class must first declare the interface in an implements clause, then provide an implementation for all the abstract methods of the interface
- A class can implement more than one interface



Extending Interfaces

- Interfaces can have sub-interfaces, just like classes have subclasses
- A sub-interface inherits all the abstract methods and constants of its super-interface and may define new abstract methods and constants

Method-Overriding

- When a class has a method of the same name, return type and by the number, type and position of its arguments as a method in its superclass, the method in the class overrides the method in the superclass
- If a method is invoked for an object of the class, its new definition of the method is called, not the superclass' old definition.

Polymorphism

- An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is usually called polymorphism

Method Overloading

- Defining methods with the same name and different argument or return types is called method overloading

For example,

```
double add(int, int)
double add(int, double)
double add(float, int)
double add(int, int, int)
double add(int, double, int)
```

Data Hiding and Encapsulation

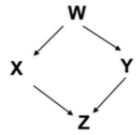
	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Constructors

- Good practice to define the required constructors for all classes
- If a constructor is not defined:
    - A no-argument constructor is implicitly inserted
    - This no-argument constructor invokes the superclass's no argument constructor
    - If the parent class (superclass) doesn't have a visible constructor with no-argument it results in a compilation error.
- If the first statement in a constructor is not super() or this(), a call to super() is implicitly inserted.
- If a constructor is define with one or more arguments no-argument constructor is not inserted in that class
- A class can have multiple constructors, with different signatures
- The word 'this' can be used to call another constructor in the same class

Diamond Inheritance Problem: A Possible Solution

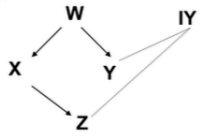
Using **multiple inheritance** (in C++):



we achieve the following:

- In class Z, we can use methods and variables defined in X, W **and** Y.
- Objects of classes Z and Y can be assigned to variables of **type Y.**
- and more ...

Using **single inheritance** in Java:




```
class W { }
interface IY { }
class X extends W { }
class Y extends W implements IY { }
class Z extends X implements IY { }
```

we achieve the following:

- In class Z, we can use methods and variables defined in X and W. In class Z, if we want to use methods implemented in class Y, we can use ***method forwarding*** technique. That means, in class Z, we can create an object of type class Y, and via this object we can access (in class Z) all the methods defined in class Y.
- Objects of classes Z and Y can be assigned to variables of **type IY** (instead of Y).
- and more ....

- Place methods of Y in IY.

2.4: Domain Modelling

Domain Models

- Domain models are used to visually represent important domain concepts and relationships between them
- Domain models help clarify and communicate important domain specific concepts and are used during the requirements of the designing phase
- Domain modelling is the activity of expressing related domain concepts into a domain model
- They are also referred to as conceptual models or domain object models
- We use UML (Unified Modelling Language) to represent domain models.

Requirements Analysis vs Domain Modelling

- Requirements analysis determines the external behaviour
- Domain modelling determines the internal behaviour
- Requirements analysis and domain modelling are mutually dependent – domain modelling supports clarification of requirements, whereas requirements help build up the model.

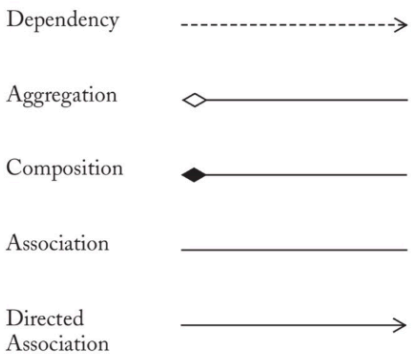
Noun/Verb Analysis

- Nouns are possible entities in the domain model
- Verbs are possible behaviours

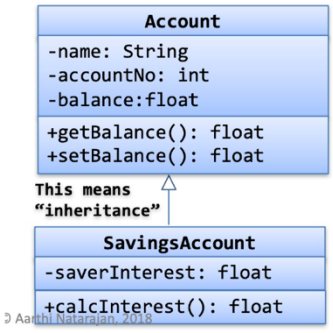
UML Class Diagrams

- Classes

- Relationships



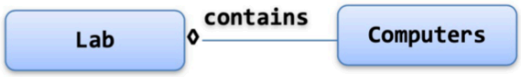
- Dependency is the loosest form of relationship
- Association is when a class 'uses' another class in some way
- When undirected, it is not clear yet the relationship
- Aggregation: A class contains another class (e.g. a course contains students)
- Composition: A class is integral to another class (e.g. leg of a chair)



Association

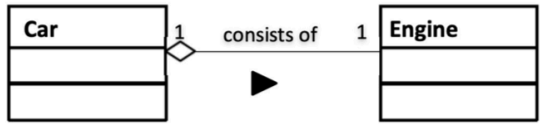
- Associations can model a 'has-a' relationship where one class contains another class
- It can be further refined as:

  Aggregation relationship (hollow diamond symbol ◊): The contained item is an element of a collection but it can also exist on its own, e.g., a lecturer in a university or a student at a university



Composition relationship (filled diamond symbol ♦ in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk, or engine in a car

- Diamond goes to the bigger thing (container)

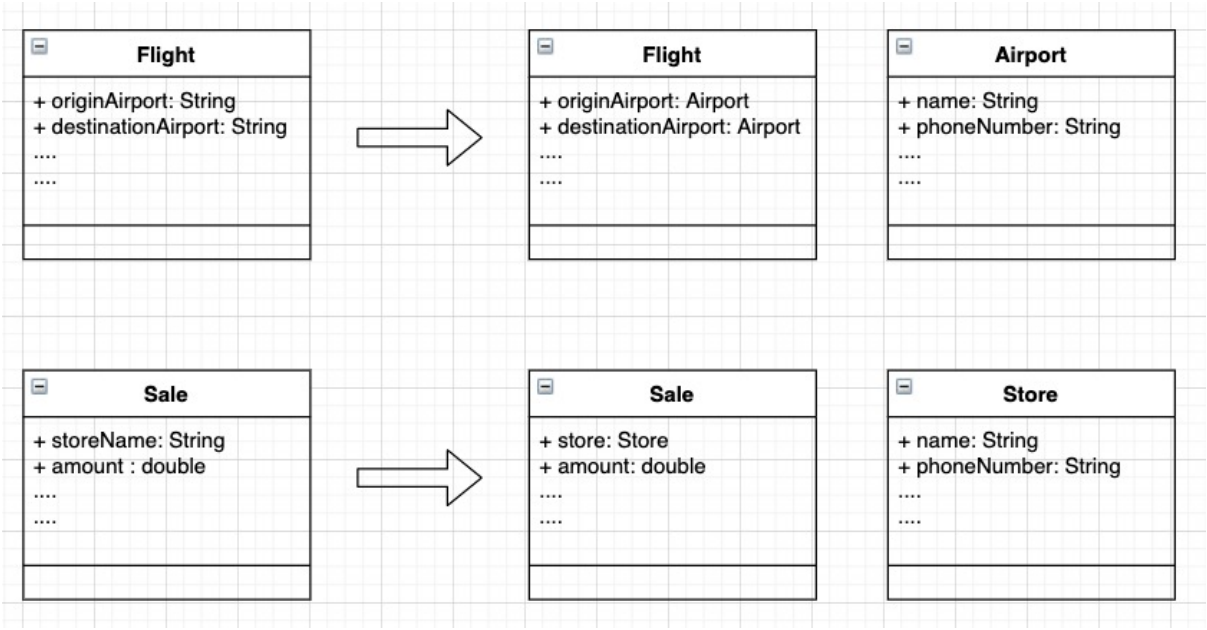


- Aggregation - "has-a" relationship where the part can exist without container

- Composition – "is-composed-of" relationship where part cannot live without container

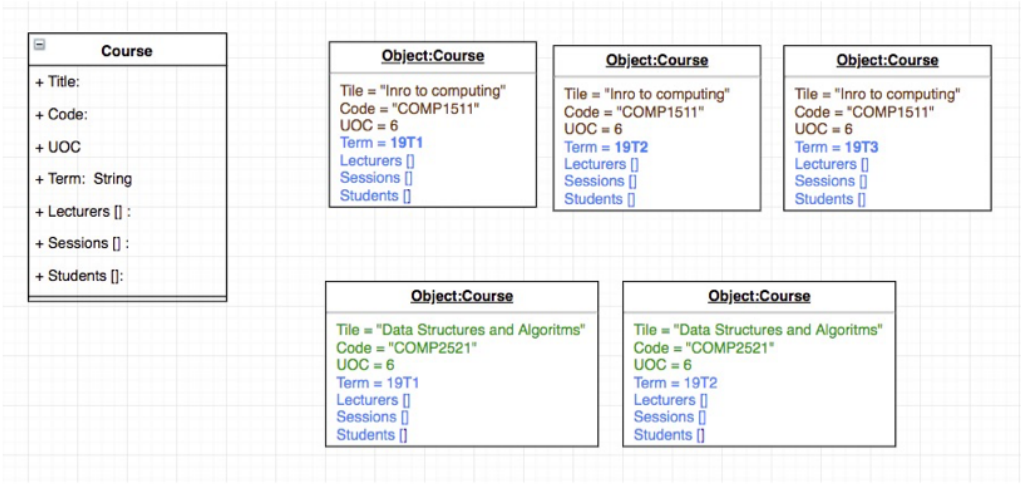
- Inheritance – "is-a-kind-of" relationship

Attributes Vs. Classes

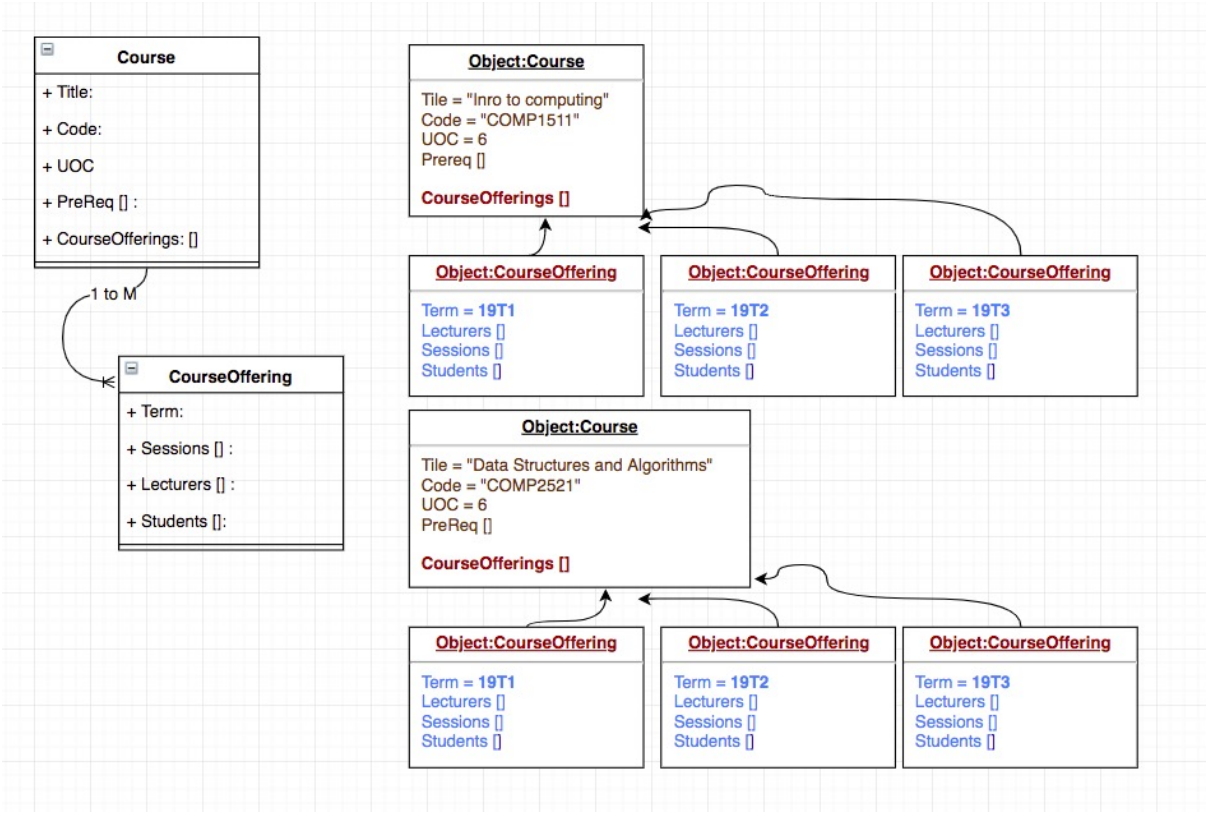


Example

- What is wrong with below?



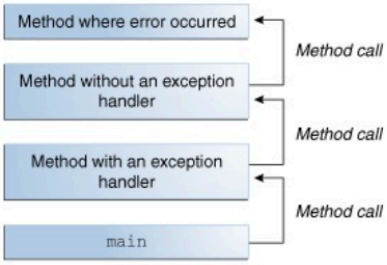
- A solution:


Course
+ Title:
+ Code:
+ UOC
+ PreReq [] :
+ CourseOfferings: []
1 to M
CourseOffering
+ Term:
+ Sessions [] :
+ Lecturers [] :
+ Students []:
Object:Course
Tile = "Inro to computing"
Code = "COMP1511"
UOC = 6
Prereq []
CourseOfferings []
Object:CourseOffering
Term = 19T1
Lecturers []
Sessions []
Students []
Object:CourseOffering
Term = 19T2
Lecturers []
Sessions []
Students []
Object:CourseOffering
Term = 19T3
Lecturers []
Sessions []
Students []
Object:Course
Tile = "Data Structures and Algorithms"
Code = "COMP2521"
UOC = 6
PreReq []
CourseOfferings []
Object:CourseOffering
Term = 19T1
Lecturers []
Sessions []
Students []
Object:CourseOffering
Term = 19T2
Lecturers []
Sessions []
Students []
Object:CourseOffering
Term = 19T3
Lecturers []
Sessions []
Students []

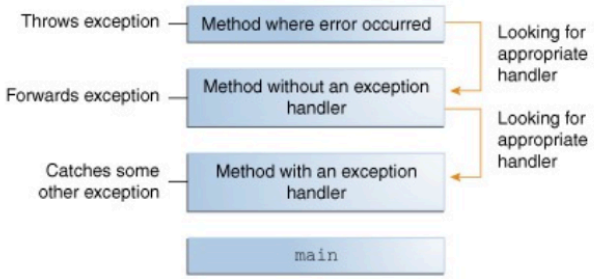
2.5: Exceptions in Java

Exceptions

- An event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions
- When an error occurs, an exception object is created and given to the runtime system, this is called throwing an exception
- The runtime system searches the call stack for a method that contains a block of code that can handle the exception
- The exception handler chosen is said to catch the exception



The call stack.



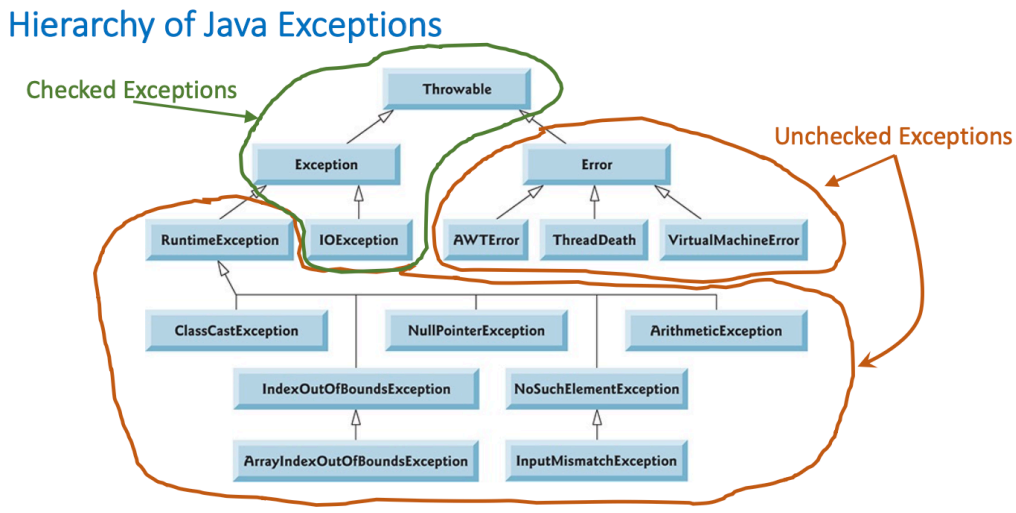
Searching the call stack for the exception handler.

Types of Exceptions in Java

- Checked exception (IOException, SQLException, etc)
- Error (VirtualMachineError, OutOfMemoryError, etc)
- Runtime exception (ArrayIndexOutOfBoundsExceptions, ArithmeticException, etc)

Checked vs Unchecked Exceptions

- All classes that are subclasses of RuntimeExcpetion (typicalled caused by defects in your code), or Error (typically 'system' issues) are unchecked exceptions
- All classes that inherit from Exception but not directly, or indirectly from class RuntimeException are considered to be checked exceptions.



Example

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());

    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());

    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

try

catch

finally

User Defined Exceptions

- We can create our own exceptions
- They must be a child of Throwable
- A checked exception needs to extend the Exception class, but not directly or indirectly from RuntimeException
- An unchecked exception needs to extend the RuntimeException class

Normally we define a *checked* exception, by extending the Exception class.

```
class MyException extends Exception {

    public MyException(String message){

        super( message );

    }

}
```

Example:

```
try {
        out = new PrintWriter(new FileWriter("myData.txt"));
        for(int i=0; i<SIZE; i++){

                int idx = i + 5;

                if(idx >= SIZE){
                        throw new MyException("idx is out of index range!");
                }
                out.println(list.get(idx));
        }

}
catch(IOException e){
        System.out.println(" In writeln ....");
}
catch(MyException e){
        System.out.println(e.getMessage());
}
catch(Exception e){
        System.out.println(" In writeln, Exception  ....");

}
```


Assertions in Java

- An assertion is a statement in Java that enables you to test your assumptions about your program
- Should not use assertions for:
    - Argument checking in public methods
    - To do any work that your application requires for correct operation
- Evaluating assertions should not result in side effects

Important: for backward compatibility, by **default,** Java **disables** assertion validation feature. It needs to be explicitly **enabled** using the following command line argument:

- `-enableassertions` command line argument, or
- `-ea` command line argument

- Example:

```
/**
 * Sets the refresh interval (which must correspond to a legal frame rate).
 *
 * @param  interval refresh interval in milliseconds.
*/
 private void setRefreshInterval(int interval) {
  // Confirm adherence to precondition in nonpublic method
  assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;

  ... // Set the refresh interval
 }
```

Exceptions: Summary Points

- Consider your exception-handling and error-recovery strategy in the design process.
- Sometimes you can prevent an exception by validating data first.
- If an exception can be handled meaningfully in a method, the method should catch the exception rather than declare it.
- If a subclass method overrides a superclass method, a subclass's *throws* clause can contain a subset of a superclass's *throws* clause. It must not throw more exceptions!
- Programmers should handle checked exceptions.
- If unchecked exceptions are expected, you must handle them gracefully.
- Only the first matching catch is executed, so select your catching class(es) carefully.
- Exceptions are part of an API documentation and contract.
- Assertions can be used to check preconditions, post-conditions and invariants.

3.1: Generics and Collections in Java

Generics in Java

- Generics enable types (classes and interfaces) to be parameters when defining:
  - Classes,
  - Interfaces and
  - Methods.
- Benefits:
  - Removes casting and offers stronger type checks at compile time.
  - Allows implementations of generic algorithms, that work on different types can be customised and are type safe.
  - Adds stability to your code by making more of your bugs detectable at compile time.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Without Generics

```
List<String> listG = new ArrayList<String>();
listG.add("hello");
String sg = listG.get(0);    // no cast
```

With Generics

Generic Types

- A generic type is a generic class or interface that is parameterised over types
- A generic class is define with the following format:

  class name< T1, T2, ..., Tn > { /* ... */ }

- The most commonly used parameter name types are:
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

For example,

```
Box<Integer> integerBox = new Box<Integer>();
```

OR

```
Box<Integer> integerBox = new Box<>();
```

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Multiple Type Parameters

- A generic class can have multiple type parameters.
- For example, the generic OrderedPair class, which implements the generic Pair interface

```java
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Generic Methods

- Generic methods introduce their own type parameters.

```java
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

The complete syntax for invoking this method would be:

```java
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean  same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown above.
Generally, this can be left out and the compiler will **infer** the **type** that is needed:

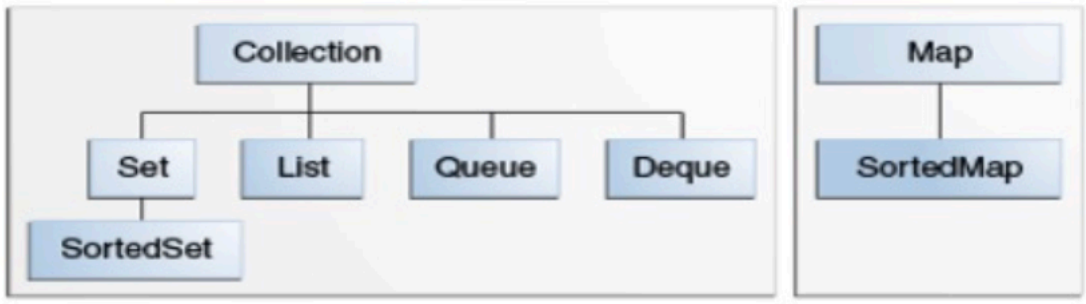
```java
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean  same = Util.compare(p1, p2);
```

Collections in Java

- Collections framework is a unified architecture for representing and manipulating collections. A collection is simply an object that groups multiple elements into a single unit.
- All collections frameworks will contain the following:
    - Interfaces: Allows collections to be manipulated independently of the details of their representation
    - Implementations: Concrete implementations of the collection interfaces
    - Algorithms: The methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces

Core Collection Interfaces

- The core collection interfaces encapsulate different types of collections
- The interfaces allow collections to be manipulated independently of the details of their representation.



The core collection interfaces.

The Collection Interface

- A collection represents a group of objects known as its elements
- The collection interface is used to pass around collections of objects where maximum generality is desired.

The Collection interface contains methods that perform basic operations, such as

- int **size**(),
- boolean **isEmpty**(),
- boolean **contains**(Object element),
- boolean **add**(E element),
- boolean **remove**(Object element),
- Iterator<E> **iterator**(),
- ... ... **many more** ...

Collection Implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Implemented Classes in the Java Collection, Read their APIs.

3.2: Design Principles

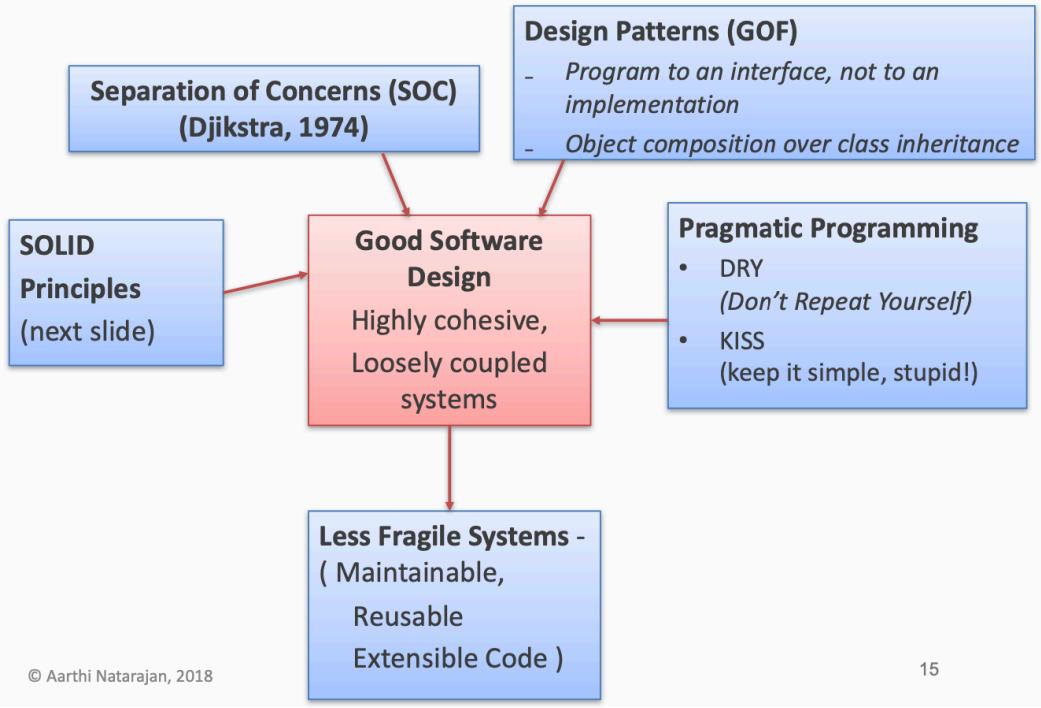
Design Smells

- Rigidity:
  - Tendency of the software being difficult to change even in simple ways
  - A single change causes a cascade of changes to the other dependent modules
- Fragility:
  - Tendency of the software to break in many places when a single change is made
- Immobility:
  - Design is hard to reuse
  - Design has parts that could be useful to other systems, but the effort needed and risk in disentangling the system is too high
- Viscosity:
  - Software viscosity – changes are easier to implement through 'hacks' over 'design preserving methods'
  - Environment viscosity – development environment is slow and inefficient
- Opacity:
  - Tendency of a module to be difficult to understand
  - Code must be written in a clear and expressive manner
- Needless complexity:
  - Contains constructs that are not currently useful
  - Developers ahead of requirements
- Needless repetition:
  - Design contains repeated structures that could potentially be unified under a single abstraction
  - Bugs found in repeated units must fixed in every repetition

Good Design

- Loose coupling: Reduce interdependence between components. Zero coupled classes are not usable!
- High cohesion: Increase degree in which all elements of a component or class or module work together as a functional unit.
- Ensures components are:
  - Extensible
  - Reusable
  - Maintainable
  - Understandable
  - Testable

Several Design Principles…One Goal



SOLID

- Single responsibility principle: A class should only have a single responsibility.
- Open–closed principle: Software entities should be open for extension, but closed for modification.
- Liskov substitution principle: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- Interface segregation principle: Many client-specific interfaces are better than one general-purpose interface.
- Dependency inversion principle: One should "depend upon abstractions, [not] concretions."

Design Principle #1

- The Principle of Least Knowledge (Law of Demeter) – Talk only to your friends
- Classes should know and interface with as few classes as possible
- Reduce interaction between objects to just a few close friends, i.e. local objects
- Helps us design a loosely coupled system.

- A method in a object should only invoke methods of:
  - Itself
  - The object passed in as a aparameter to the method
  - Objects instantiated within the method
  - Any component objects
  - Not those of objects returned by a method
- Do not dig deep inside your friends of friends of friends of friends and get in deep conversations with them, do not do o.get(name).get(thing).remove(node)

- Rule 1:

A method **M** in an object **O** can call on any other method within **O** itself

```
public class M {
 public void methodM() {
        this.methodN();
 }
 public void methodN() {
        // do something
 }
}
```

- Rule 2:

A method **M** in an object **O** can call on any methods of parameters passed to the method **M**

```
public class O {

public void M(Friend f) {
// Invoking a method on a parameter passed to the method is
// legal
f.N();
}

public class Friend {

 public void N() {
 // do something
 }
}
```

- Rule 3:

A method **M** can call a method **N** of another object, if that object is instantiated within the method **M**

```
public class O {

   public void M() {
        Friend f = new Friend();
        // Invoking a method on an object created within the
        // method is legal
        f.N();
   }

public class Friend {
        public void N() {
         // do something
        }
}
```

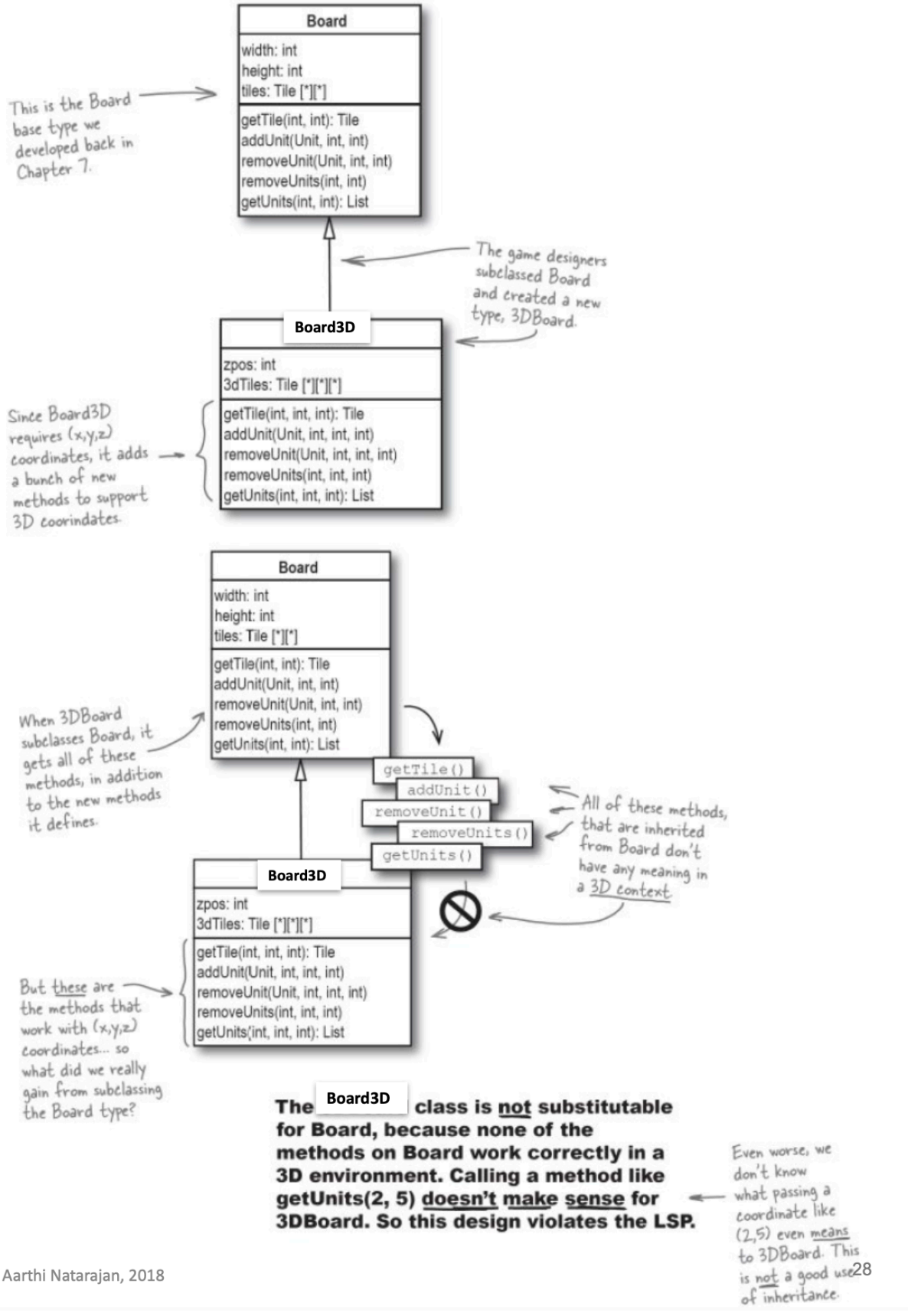

- Rule 4:

Any method **M** in an object **O** can call on any methods of any type of object that is a direct component of **O**

```
public class O {

   public Friend instanceVar = new Friend();

   public void M4() {
   // Any method can access the methods of the friend class
   F through the instance variable "instanceVar"
   instanceVar.N();
   }
public class Friend {
        public void N() {
         // do something
        }
}
```

Design Principle #2

- LSP (Liskov Substitution Principle): Subtypes must be substitutable for their base types
- Example:



LSP states that subtypes must be substitutable for their base types

```
Board board = new Board3D()
```

But, when you start to *use* the instance of `Board3D` like a `Board`, things go wrong

```
Artillery unit = board.getUnits(8,4)
```

Board here is actually an instance of the sub-type Board3D

But, what does this method for a 3D board?

Inheritance and LSP indicate that any method on `Board` should be able to use on a `Board3D`, and that `Board3D` can stand in for `Board` without any problems, so the above example clearly violates LSP

Solve problems without inheritance

- Delegation – delegate the functionality to another class
- Composition – reuse behaviour using one or more classes with composition
- Design principle: Favour composition or delegation over inheritance

Method Overriding

- The argument list should be exactly the same as of the overridden method
- The access level cannot be more restrictive than the overridden method's access level
- A method declared final cannot be overridden
- Constructors cannot be overridden
- Rules:
    - The return type in the overridden method should be the same or a sub-type of the return type defined in the super-class
    - This means that return types in the overridden method may be narrower than the parent return types

```
public class AnimalShelter {

    public Animal getAnimalForAdoption() {

        return null;
    }

    public void putAnimal(Animal someAnimal){

    }

}
```

```
public class CatShelter extends AnimalShelter {

    /*
     * @see AnimalShelter#getAnimalForAdoption()
     */
    @Override
    public Cat getAnimalForAdoption() {

            //Returning a narrower type than parent
            return new Cat();
    }

}
```

3.3: JUnit

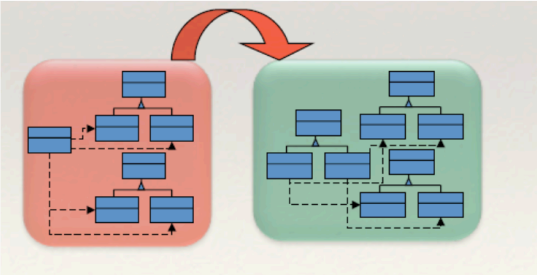
Example:

```java
/**
 * Tests for Pineapple on Piazza
 * @author Nick Patrikeos
 */
public class PiazzaTest {

    @Test
    public void testExampleUsage() {
        // Create a forum and make some posts!
        PiazzaForum forum = new PiazzaForum("COMP2511");
        assertEquals("COMP2511", forum.getName());

        Thread funThread = forum.publish("The Real Question - Pineapple on Piazza", "Who likes pineapple on piazza?");

        funThread.setTags(new String[] { "pizza", "coding", "social", "hobbies" });
        assertTrue(
                Arrays.equals(new String[] { "coding", "hobbies", "pizza", "social" }, funThread.getTags().toArray()));

        funThread.publishPost("Yuck!");
        funThread.publishPost("Yes, pineapple on pizza is the absolute best");
        funThread.publishPost("I think you misspelled pizza btw");
        funThread.publishPost("I'll just fix that lol");

        assertEquals(5, funThread.getPosts().size());
    }

    @Test
    public void testSearchByTag() {
        PiazzaForum forum = new PiazzaForum("COMP2511");

        Thread labThread = forum.publish("Lab 01", "How do I do the piazza exercise?");
        Thread assignmentThread = forum.publish("Assignment", "Are we back in blackout?");
        labThread.setTags(new String[] { "Java" });
        assignmentThread.setTags(new String[] { "Java" });

        List<Thread> searchResults = forum.searchByTag("Java");
        assertEquals("Lab 01", searchResults.get(0).getTitle());
        assertEquals("Assignment", searchResults.get(1).getTitle());
    }
}
```

```java
public class ArchaicFsTest {
    @Test
    public void testCdInvalidDirectory() {
        ArchaicFileSystem fs = new ArchaicFileSystem();

        // Try to change directory to an invalid one
        assertThrows(UNSWNoSuchFileException.class, () -> {
            fs.cd("/usr/bin/cool-stuff");
        });
    }

    @Test
    public void testCdValidDirectory() {
        ArchaicFileSystem fs = new ArchaicFileSystem();

        assertDoesNotThrow(() -> {
            fs.mkdir("/usr/bin/cool-stuff", true, false);
            fs.cd("/usr/bin/cool-stuff");
        });
    }

    @Test
    public void testCdAroundPaths() {
        ArchaicFileSystem fs = new ArchaicFileSystem();

        assertDoesNotThrow(() -> {
            fs.mkdir("/usr/bin/cool-stuff", true, false);
            fs.cd("/usr/bin/cool-stuff");
            assertEquals("/usr/bin/cool-stuff", fs.cwd());
            fs.cd("..");
            assertEquals("/usr/bin", fs.cwd());
            fs.cd("../bin/..");
            assertEquals("/usr", fs.cwd());
        });
    }

    @Test
    public void testCreateFile() {
        ArchaicFileSystem fs = new ArchaicFileSystem();

        assertDoesNotThrow(() -> {
            fs.writeToFile("test.txt", "My Content", EnumSet.of(FileWriteOptions.CREATE, FileWriteOptions.TRUNCATE));
            assertEquals("My Content", fs.readFromFile("test.txt"));
            fs.writeToFile("test.txt", "New Content", EnumSet.of(FileWriteOptions.TRUNCATE));
            assertEquals("New Content", fs.readFromFile("test.txt"));
        });
    }
```

4.1: Refactoring

The process of restructuring (changing the internal structure of software) software to make it *easier to understand* and *cheaper to modify* without changing its *external, observable behaviour*



- Refactoring improves design of software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster
- Refactoring helps you to conform to design principles and avoid design smells

When should you refactor?

Tip: *When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature*

Refactor when:

- You add a function (swap hats between adding a function and refactoring)
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

Common Bad Code Smells

- Duplicate code
- Long method
- Large class
- Long parameter list
- Divergent change (when one class is commonly changed in different ways for different reasons)
- Shotgun surgery (when you make a lot of little changes to a lot of different classes)