FIT2099
Object Oriented Design & Implementation

Design Principles

Topics Covered
Object Oriented Paradigm
Encapsulation
Abstraction
Code Smells and Refactoring
Other Principles

Object Oriented Paradigm
Object orientation is a conceptualisation of objects that carry out the programs tasks.

Procedural Programming
- A collection of procedures
- Each procedure has its own input and output
- Easier to write spaghetti code

Object oriented programming flips this around

Object Oriented Programming
- Unit of organisation is the object
- Objects are instances of classes
- A class defines an interface

Encapsulation
Encapsulation fulfils several definitions:
1. A software development technique that consists of isolating a system function or set of data and operations on the data within a module and providing precise specifications for the module
2. The concept that access to the names, meanings, and values of the methods of a class is entirely separated from access to their realisation
3. The idea that a module has an outside that is distinct from its inside, the it has an external interface and an internal implementation

Information Hiding
- Information/Implementation hiding is the use of encapsulation to restrict from external visibility certain information or implementation decisions that are internal to the encapsulation structure.

Mechanisms for Encapsulation
Java was made to encapsulate. The basic unit of Java programs is the class. Java can restrict access to things in the class as:
- Within the class only (private)
- Within the package (default/unspecified)
- Only to subclasses (protected)
- No restriction (public)

Encapsulation Boundaries
An encapsulation boundary is something across which visibility can be restricted.
Any calls to methods not in a class crosses an encapsulation boundary; these need to be minimised.
To enforce encapsulation, you can do the following
- Avoid public attributes
- Only make the methods public when necessary
- Keep the class package-private if not needed
- Use protected sparingly
- Minimise interfaces
- Defensively copy when using getters to eliminate effects from mutability.
  - Avoid privacy leaks
- Don't expose implementation details, avoid returning the copy in the original data type when a better one can be used
- Be aware and avoid relying on algorithm quirks (eg, when data is returned sorted, incidentally)

Abstraction
Abstraction is the act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances.
As a developer, this means deciding
- What information we need in order to represent an item or object
- What we should expose to use this part easily

We use abstraction when we bundle things together and use them, eg lines of code in a method, data in a class, classes in a package.

Abstraction in OO Design
We want to design our own software in such a way as to make it easier to maintain, extend, and modify.
If we make developers lives easier, we will produce software more effectively:
- Accrue less technical debt
- Make iterative development easier
- Respond more readily to changes in requirements or environment
- Reduce cognitive load on a developer

Abstraction v Encapsulation

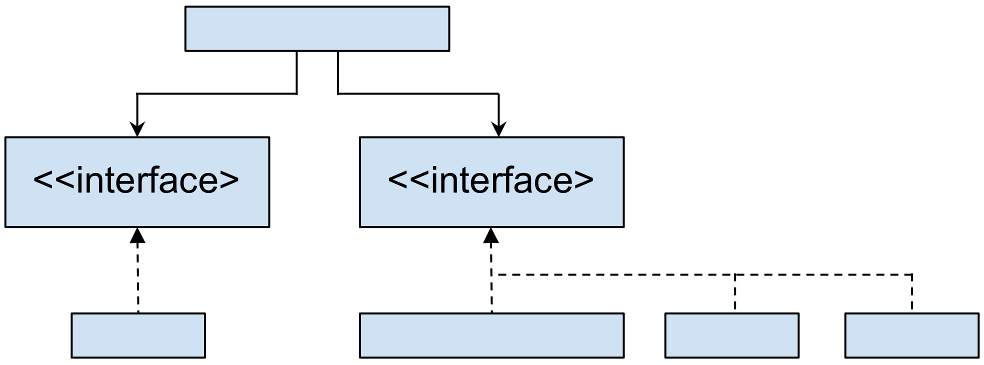
- We use encapsulation when we bundle things together
- We use abstraction when we decide what to bundle together
    - Or how things should look from the outside
- We use information hiding when we use encapsulation that doesn't allow access from the outside

Separation of Concerns

A principle for separating a program into sections, or modules with their own concern or responsibilities. Concerns should be well defined and have little overlap with others.

The Dependency Inversion Principle (DIP)

High level modules shouldn't depend on low level modules, but rather depend on abstractions.
For example, consider the following which uses interfaces to abstract different implementations



Common mistakes with abstraction

1. Misunderstanding simplicity
    a. Confusing simple design with fewest possible classes
        i. One way to reduce complexity is define smaller classes
        ii. Minimize dependencies between classes
2. Overdoing dependency inversion
    a. Using too much abstraction is also bad
3. Confusing abstraction with abstract

Using Abstraction in Java

Abstraction is a design principle rather than a programming technique, but most languages support it in

- **Classes:** The class is the most important mechanism in abstraction; A well designed class should represent a single concept, expose a public interface for its responsibility, and hide implementation that doesn't fulfil that responsibility
  - *Visibility Modifiers*: Deciding what to hide and expose is a big part of applying abstraction
  - *Abstract Classes*: A subclass inherits all the non-private methods declared in a base class. Client code can be passed an instance of some concrete subclass without needing to know its type. All it knows is that it does everything the subclass can
  - *Hinge Points*: Applying dependency inversion to a simple relationship; client code doesn't care about implementation and vice versa, the parts can move around freely except for where they're joined.
- **Packages**
  - We don't want our classes to be too large but we may need a lot of code to implement a feature. The solution is packages
    - Group related classes into a subsystem
    - Come up with a name
    - Put the package name at the top of each class
    - Move the Java files into a directory with the package name
    - *Nesting Packages*: You can't place a package in a package, but you can use dot notation to group packages together

Abstraction Layers

An abstraction layer is the publicly accessible interface to a class, package or subsystem.
You can create an abstraction layer by restricting visibility as much as possible

- Ideally make everything private
- If not private, then package
- Use hinges to avoid public

Interfaces

Used exclusively in Java

- Separate publicly accessible interface from their implementations
- Can be seen as an extreme abstract class

Generics

Generics allow us to define a class that may require varying types of data types. For examples, without generics we couldn't create a list of both strings and integers, but with generics we can use <> notation to specify the types we want to use

```
For example:
    List<int, float> = new ArrayList<>();
```

Bound type parameters

Write a generic collection class that places a restriction on the classes of objects it can store.

```
public class BinarySearchTree<T extends Comparable<T>> { }
```

Code Smells and Refactoring

Code Smells

An experienced developer develops the ability to detect bad design and implementation almost automatically when viewing code. This is through the identification of problems within the code. Ie, it's a surface indication that corresponds to a deeper problem in the system.

Refactoring

A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.
Refactoring can improve design, understandability and makes debugging easier.
Why refactor?

- Improve design(extensibility)
- Improve understandability
- Makes debugging easier

You should refactor when:

- When adding new features
- When you need to fix a bug
- As you do a code review

Don't refactor and add new features at the same time

Taxonomy of Code Smells

- **Long Smells**
    - Duplicated Code
        - Don't Repeat yourself
    - Long Methods
        - Difficult to understand
    - Large Classes
        - Violates single responsibility principle and low cohesion
    - Long Parameter List
        - Should have the data it needs in the class
        - More than three is usually bad
- **Social Smells**
    - Divergent Change
        - You have a class that repeatedly changes, in a couple of ways
        - You can often change several methods together
    - Shotgun Surgery
        - The opposite of divergent change - when adding functionality, you end up changing many classes
        - Indicative of poor encapsulation
    - Feature Envy
        - A method spends its effort calling another class

- **Smells Like Python**
    - Primitive Obsession
        - Storing everything in primitives rather than classes
        - Makes validation difficult
    - Data Clumps
        - Different variables that represent the same information
    - Switch Statements
        - If you want to change or extend it, you have to find and change them all
        - Polymorphism is the answer
    - Data classes
        - Classes with data and no logic
- **Couplers**
    - Message Chains
        - A message chain occurs when a client requests another object, that object requests yet another one, and so on.
        - `X = getThingy().getStuff().getAgain()`
    - Middle man
        - If a class performs only one action, delegating work to another class, why does it exist at all?
- **Overengineering**
    - Speculative Generality
        - When you add methods for every special case
    - Lazy Class
        - Class doesn't do much anymore after refactoring

Refactor Methods

- **Extract Method**
    a. Create a new method named after it's intention
    b. Figure out visibility
    c. Copy the extracted code from the source method into the new target method
    d. Sort out issues with local variables
    e. Insert a call to method

Refactoring with Fowler

- Long Method
    - Fix long methods by extracting parts
- **Temporary Variables**
    - Fowler doesn't like them:
        - Only useful in their own routine, Easy to lose track of
    - Replace them with queries
        - Accessible to any method
        - Exchanges cleaner design

Other Principles

Separation of Concerns

A principle for separating a program into sections, or modules with their own concern or responsibilities. Concerns should be well defined and have little overlap with others.
See Abstraction

The Dependency Inversion Principle (DIP)

High level modules shouldn't depend on low level modules, but rather depend on abstractions.
See Abstraction

Single Responsibility Principle

every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.
See Code Smells

Liskov Substitution Principle

- For a class S to be a true subtype of T, then S must conform to T
- A class S conforms to class T only if an object of class S can be provided in any contract where an object of class T is expected and correctness is still preserved

See Design By Contract

Command-Query Principle

Every method should either be a command or query

- A command performs actions changing the states of objects
- A query returns a value with no side effects

See Design By Contract

Design Methodologies

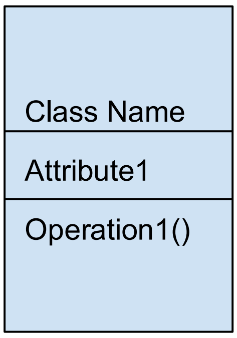
Topics Covered
Unified Modelling Language (UML)
Software Specifications and Design
Design Generation Methodologies
JavaDocs
Design by Contract and Conditions
When and Why Design
Development Methodologies

Unified Modelling Language (UML)

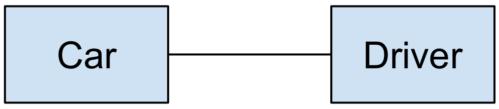
Class Diagrams

A class is a set of objects that share attributes and/or operations. There is a correspondence between class diagrams in UML and ER diagrams. Classes are analogous to entities, the only difference being the lack of operations in entities.

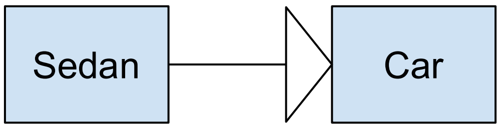
Class Diagram Notation



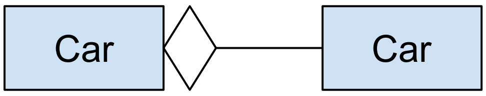
- **Classes**
  - Operations and attributes can be omitted to emphasise other elements
- **Relationships**
  - **Association**



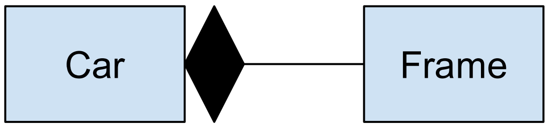
    - Allows one object to perform an action on its behalf
    - May also have an arrow between the two to indicate that only one knows about the other
  - **Generalisation/Inheritance**



    - One of the classes is a specialised form of the other



- **Aggregation**
  - One class is made up of the other, but one will continue to exist once the other is gone
  - Eg. a university has departments, and they have professors. If the uni closes the departments no longer exist, but the professors do
- **Composition**



  - One object is part of the other

Association Classes
Keeps track of information about the association itself and to add attributes, operations and other features to associations

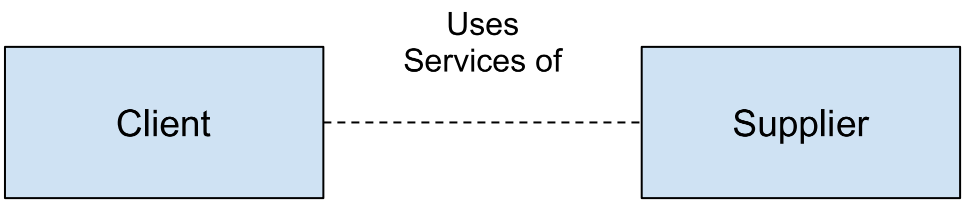
Dependencies
A dependency is a relationship which indicates that a change in specification of one thing may affect another thing it uses. You should use dependency when you want to show one thing using another, but there isn't an association. This is shown as a dotted line

Constraints
A specific constraint on a system to ensure a robust system. This is represented by a description of the constraint within curly brackets along an association.

Stereotypes
A stereotype tells you something interesting about a system. It is represented by two <<arrows>>

Software Specifications and Design

Client/Supplier Relationship
In UML, this is shown as an association on a dependency.



Design Generation Methodologies

- **Brainstorming**
  - Go for quantity
  - Without criticism
  - Welcome wild ideas
  - Combine and improve elements
  - Throw out chaff later
- **Bottom Up**
  - Start with a small problem
  - Design a solution to that
  - Do a few more
  - Put them together for a solution
- **Top Down**
  - Start with a high-level problem
  - Divide into sub problems
  - Solve these and put it together
- **Scenario Based**
  - Have scenarios that the design need to support
  - Work through the scenarios
  - Improve design to support scenarios more effectively
- **Class Responsibility Collaboration (CRC) Cards**
  - No special notation needed
  - We start with obvious cards and start playing ‘what if’ with scenarios. If the situation calls for a new responsibility, either:
    - Add a new responsibility
    - Create a new object
  - Add collaborations as we go
  - CRC helps with encapsulation, it encourages small objects with clear responsibilities. Though it doesn’t generate good design, and this needs to be kept in mind.

Class Name	
Responsibilities ↓	Collaborators ↓

JavaDocs
JavaDoc is a documentation generator in Java that converts the documentation given to packages, classes, methods, and other attributes into a HTML format.
JavaDocs separates implementation from interface, and therefore, allows code to be used without the source code.
Note that Javadocs shouldn't, and needn't be attached to anything which is private, and it must occur outside of methods, classes and packages.

Formatting

```
/**
 * This is a Javadoc comment for the method <code>foo</code> and should explain what
 * <code>foo</code> does. Not that you can use HTML tags within your JavaDocs.
 * {@code return "code snippets can be include as so";
 *
 * @param FIT2099 a description of the parameter
 * @return a description of the return
 */
public String foo(String FIT2099) {
    code;
}
```

Design by Contract
A class designer establishes a software contract between him/herself and the users of the class they design. We can make this impersonal and think of this as a contract between the class (supplier) and the classes that use that class (client).
The software contract provides the documentation of the class for the technical user and the possibility of enforcing the contract by using exceptions and assertions
The software designer tells the user what the class does providing a specification for the class.
A specification:
- Is ideally part of the implementation
- Should ideally be extractable from the implementation
- Is essential for supporting component reuse and maintenance
- Is more than just the API we have gotten used to seeing

The specification forms the public interface of the class
The user:
- Should be able to know how to use a class by reading its specifications
- Should not have to look at implementation details

Exceptions and Assertions in Contracts

Contracts can be defined by using assertions and exceptions to create **executable specifications**

- They are verifiable by the compiler or code
- Go beyond simply commenting specifications

Also see preconditions, postconditions, and invariant

Precondition Violation

- The client is at fault and an exception should be thrown to the caller.
- Suppliers should not try to rescue the clients mistake.
- The calling function should handle the exception.
- Indicates a bug.

Postcondition Violation

- The supplier/designer is at fault and an exception is raised in the called routine.
- Could be caused by a bug or transient condition.
- Doesn't always indicate a bug
    - Network Outrage
    - Remote Server Down
    - Disk or Memory Unavailable

Disciplined Exception Handling Principle

- There are only two legitimate responses to an exception that occurs during the execution of a routine
    - Retrying
        - An attempt to change the condition that led to the exception and to execute the routine again from the start
        - Asking user to retry again
    - Failure
        - Clean up the environment, terminate the call and report failure to caller
        - Correct the exception in an organized manner

Non-redundancy principle

- Under no circumstances shall the body of a routine test for the routine's precondition
- Contradicts "defensive programming" principles
- It's the client's responsibility to check that it is meeting the precondition of its suppliers

Fail Fast

- The fail fast principle says that a system should fail immediately and visibly when something is wrong.
- This allows a developer to easily find a problem and fix it when it arises, rather than having it stay unknown and cause problems later.
- Design by Contract allows to write code that can fail fast.

See Debugging, Assertions, and Exceptions

Liskov Substitution Principle

- For a class S to be a true subtype of T, then S must conform to T
- A class S conforms to class T only if an object of class S can be provided in any contract where an object of class T is expected and correctness is still preserved

This leads to the design by contract paradigm with the following rule

- Subclasses must honour the contracts of their parents
- If this is done, we can use a subclass where its parent is expected

Sound Subcontracting

If a class is subcontracted by a supplier, the client doesn't need to know this.
This leads to the following rules for precondition and postconditions in a subclass:

- A subclass can only weaken the preconditions of its parents
  - Expect more from a supplier
  - New preconditions should be logically 'or'-ed with those of its parents
- A subclass can only strengthen the postconditions of its parents
  - Guarantee more to a supplier
  - New postconditions should be logically 'and'-ed with those of its parents
- A subclass must preserve invariants

Command-Query Separation Principle

Every method should either be a command or query

- A command performs actions changing the states of objects
- A query returns a value with no side effects

Development Methodologies

Waterfall

Requirements ⟶ Design ⟶ Implementation ⟶ Verification ⟶ Maintenance

Or, with slight refinement

Requirements ⟶ Analysis ⟶ Design

Typical times for conscious design

- Before implementing something big and complex
- Refactoring existing code

Technical Debt

- You have a piece of functionality that you need to add to your system You see two ways to do it, one is messy but quick - though it will cause problems in the future, the other is cleaner but slow - though it will be free of potential problems.
- This dilemma is inevitable in real systems. It happens because sometimes we need to get features out the door, though it can be repaid by refactoring

Aim in Design

- A good design
- An understanding of that design in the eyes of stakeholders
- Produce documents and designs to aid the following
    - Preliminary Domain Model
    - Sequence Diagrams
    - Revised Domain Model
    - Revised Sequence Diagram

What is a conscious, good design?

- Systematically making good decisions about how to build software
- Turing big hard problem into lot of easy small ones

What is a model?

- A representation of some aspect of the system we wish to model

What can we model?

- Structure of system(static)
- Behaviour of system(dynamic)
- Use both with feedback between each other

Two types of models

1. Static modelling
    a. Structure Diagram
        i. Profile Diagram
        ii. Class diagram
        iii. Composite Structure Diagram
        iv. Component Diagram
        v. Deployment Diagram
        vi. Object Diagram
        vii. Package Diagram
2. Dynamic Modelling
    a. Behaviour Diagram
        i. Activity Diagram
        ii. Use Case Diagram
        iii. State Machine Diagram
        iv. Interactive Diagram
            1. **Sequence Diagram**
            2. Communication Diagram
            3. Interaction Overview Diagram
            4. Timing Diagram

Dependency Control

Why Dependencies?

- Dependencies are unavoidable
- We want dependencies to be
    - Only present when necessary
    - Explicit
    - Easy to understand

Connascence

More explicitly, I define two software elements A and B to be connascent if there is at least one change that could be made to A that would necessitate a change to B in order to preserve overall correctness --Meilir Page-Jones

Types of Connascence

- Static
    - Obvious from code structure
    - Can be automatically identified by IDE/analysis tools
- Dynamic
    - Only obvious from close inspection/execution
    - Can't be easily identified by IDE
    - Generally, more concerning.

Types of Connascence

1. Connascence of Name (CoN)
    a. When you need things in two places to have the same name
2. Connascence of Type (CoT)
    a. When two things have the same type
3. Connascence of Position
4. Connascence of meaning/convention (CoM/CoC)
5. Connascence of Algorithm (CoA)
6. Connascence of Execution(CoE)
7. Connascence of Timing (CoT)
8. Connascence of values(CoV)
9. Connascence of Identity(CoI)

More Connascence means,

1. Harder to extend
2. More chance of bugs
3. Slower to write in the first place

Locality Matters,

- Within a method -> almost (but not totally) irrelevant.
- Between two methods in a class -> often no big deal.
- Two classes -> warning warning
- Two classes in different packages -> WARNING WARNING
- Across application boundaries -> keep to absolute minimum.

Contranascence

- When two things are required to be different
- Aliasing bugs – an example fault type where contrascnece is not maintained

What can we do about Connascence?

1. Minimize overall amount of Connascence by breaking system into *encapsulated* elements.
2. Minimize remaining Connascence that crosses *encapsulation boundaries*
3. Maximize Connascence *within encapsulation boundaries*

Object Oriented Concepts (Java)

Topics Covered
Java Fundamentals
Classes, interfaces, and abstract classes
Generics
Packages
Super classes
Dependencies
Debugging, Assertions, and Exceptions
Preconditions, Postconditions, and Invariants

Java Fundamentals

Primitives

- **Byte:** 8-bit integer
- **Short:** 16-bit integer
- **int:** 32-bit integer
- **Long:** 64-bit integer
- **Float:** 32-bit floating point
- **Double:** 64-bit floating point
- **Boolean:** True or False
- **Char:** 16-bit Unicode character

Variable Declaration

```
object varName = new Object(arguments);
primitive varName = primitiveValue;
```

Named Constants

Stores a variable which can only be set once

```
static final type CONSTANT_NAME;
```

Control Structures

- **if, else if, else**

```
if (condition) {
    code;
}
else if (condition) {
    code;
}
else {
    code;
}
```

- **While**

```
while (condition) {
    code;
}
```

- **For**

```
for (initialisation, condition, action) {
    code
}
```

Arrays, Lists, and Maps

A basic array contains a fixed number of items of the same type, in a fixed order

```
Object[] arrayName = new Object[n];
arrayName[i] = value;
value = arrayName[i]
```

Lists are a class that acts like an array, but is more flexible in its ability to resize when something is added

```
List<Object> listName = new List<Object>();
listName.add(value);
listName.remove(i);
value = listName.get(i);
```

Maps are a class that acts like a list, but it uses a key rather than an index to store and get values

```
Map<Object, Object> mapName = new Map<Object, Object>();
mapName.put(key, value);
value = mapName.get(key);
```

Classes, Interfaces, and Abstract Classes

Classes

A class consists of:

- A declaration
  - Visibility
    - Public
    - Protected
    - Private
  - The class it inherits from
    - Extends
  - Any interface it implements
    - Implements
- Clauses of members in the class
  - Type
  - Visibility
  - Initial values

Every object created has a unique identity, independent of the objects state.

```
visibility class ClassName extends Parent implements Interface {
    visibility Object varName = new Object(parameters);

    visibility type method(arguments) {
        code;
    }
}
```

Interfaces

Java allows us to specify a type that declares what methods any class that implements that type must have, without providing any implementation for these methods

Abstract Classes

An abstract class has at least one abstract method; that is, a method with a declaration but no implementation.
The abstract methods act as a placeholder which could be different for certain superclasses, and normal methods which are inherited identically by every superclass.

```
Abstract class in Java:
    visibility abstract class SubClassName {
        visibility type SubClassName(parameters) {
            Code;
        }
        // Abstract Method
        abstract visibility type abstractMethod(parameters)
    }

Use of abstract classes in a superclass:
    visibility class SuperClassName extends SubClassName {
        visibility type SuperClassName(parameters) {
            super(arguments);
            code;
        }
        // Use of abstract code
        @Override
        visibility type abstractMethod(parameters) {
            code;
        }
    }
```

The Universal Base Object

Every class in Java is a subclass of the Object class. This comes with a set of default methods which are automatically inherited.

```
Create and return a copy of the object:
    protected Object clone() throws CloneNotSupportedException
Indicate whether some object is equal to this one
    public boolean equals(Object obj)
Called by the garbage collected when there are no more references to an object
    protected void finalize() thrown throwable
Return the runtime class of an object
    public final Class getClass()
Returns a hash code value for the object
    public int hashCode()
Returns a string representation of the code
    public String toString()
```

All of these methods are overridable by superclasses, allowing developers to design their own implementation

Packages
A package is a group of related classes. Its benefits are as follows:
- It makes it easier to find related classes
- It can prevent name clashes
- Eliminates dependencies

Dependencies should be reduced as must as possible and elements that must depend on each other should be grouped within an encapsulation boundary. Related to this, dependencies should be minimised for those which cross boundaries and things should be declared within the tightest possible scope.

Dependencies

Dependencies
a broad software engineering term used to refer when a piece of software relies on another one.

Indirect Dependencies
Some dependencies are obvious, others are invisible to the compiler. They exist because the meaning of various elements in the code to humans; only a human can know they exist. These are the most dangerous dependencies as they're not able to be found by software and can be overlooked during maintenance.

Debugging, Assertions, and Exceptions

Fail Fast
The fail fast principle says that a system should fail immediately and visibly when something is wrong. This allows a developer to easily find a problem and fix it when it arises, rather than having it stay unknown and cause problems later.

Assertions
Java provides a mechanism for the programmer to assert something that should be at a certain point in the code.

```
assert condition : expression
```

Exceptions
Sometimes when things go wrong the method executing cannot do its job, so there needs to be a way to report the problem so that it can be fixed.
In Java, when a method needs to signal that something's gone wrong, it does so by throwing an exception. The method that called the method can either try to catch the exception and deal with it, or throw it to the method that called it.

```
Throwing exceptions:
    thrown new Exception(message);
The method that throws the exception needs to declare that it can throw the exception
    visibility type methodName(parameters) throws ExceptionName
```

There are only two ways an exception should be should be handled

- Retrying
  - To change the conditions that caused the bug
- Failure
  - Clean the environment, terminate the call, and report the failure to the caller

There are four main reasons for using exceptions and assertions

- Help in writing correct software
- Aid in documentation
- Support for testing
- Support for software fault tolerance

They are not

- An input checking mechanism
- Control structure

Catching Exceptions

Rather than passing an exception on, the calling method has the option of trying to deal with it. We do this by catching the exception.

```
try {
    code;
    }
catch(ExceptionName e) {
    handle e;
    }
```

Unchecked Exception

These are superclasses of `RuntimeException` or `Error`, and they do not have to be specified in the signature of a method. They are used in situations in which it isn't reasonable for the client to be able to recover or handle the exception.

Preconditions, Postconditions, and Invariants

- **Preconditions**
  - What the client must guarantee to do
  - Usually in the form of constraints on the arguments of a method
  - Violation of a precondition indicates a bug
- **Postconditions**
  - What the supplier must guarantee to provide
  - Violation of a post condition is usually due to a bug
- **Invariants**
  - Conditions that are held throughout the class

Redundancy

In some cases, the implementation of a routine and it's postconditions are similar. Often, we can, and should write the preconditions before we implement any code. This leads to the non-redundancy principle; under no circumstances shall the body of a routine test for the routine's precondition

Difference between checked and unchecked exception

Checked exception are detected and thrown by the compiler during compilation.

How to handle?

- use *try/catch* to enclose the code that is bound to throw compile time exception. OR
- declare the exception using *throws* keyword with method signature.

Example: IOException, SQLException, FileNotFoundException, etc

Unchecked Exception are those that escapes the notice of compiler. It is generally error in programming logic.

Like division by zero or trying to access a value from index which does not exists.
This exception will be caught during ***run time***.
How to handle?
You have to be sure as a programmer that your coding logic is accurate. You cannot write a code like
int i=5, int j=0;int k=i/j; //**logical error (ArithmeticException)**
If you do, you have to debug the code and find the exact logical error and fix at runtime.