Git. SOLID. Collections. Exceptions

Agenda

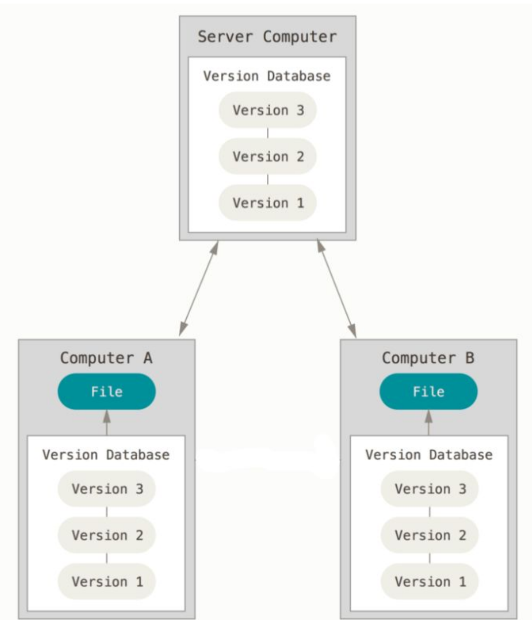
1) What is Git and how to use it?
2) What is SOLID about?
3) What collections do exists in standard Java library and how to use them?
4) How to deal with exceptional situations?

Git - the stupid content tracker

Git - Distributed Version Control System

Unlike a centralized system, clients don’t just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history

Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

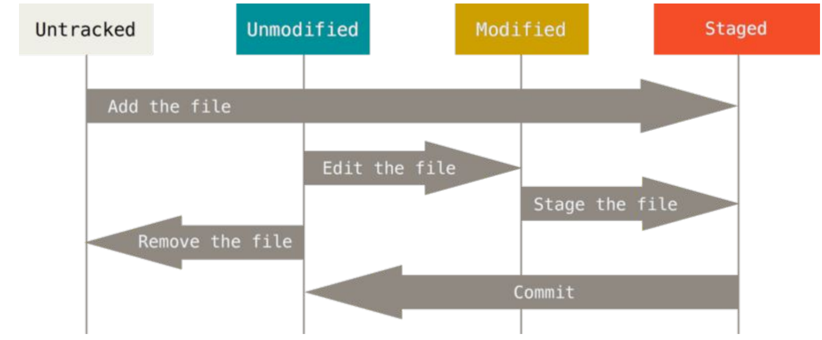


Recording changes

Each file in your working directory can be in one of two states: **tracked** or **untracked**.

Tracked files are files that were in the last snapshot, as well as any newly staged files; they can be **unmodified**, **modified**, or **staged**. In short, tracked files are files that **Git knows about**.

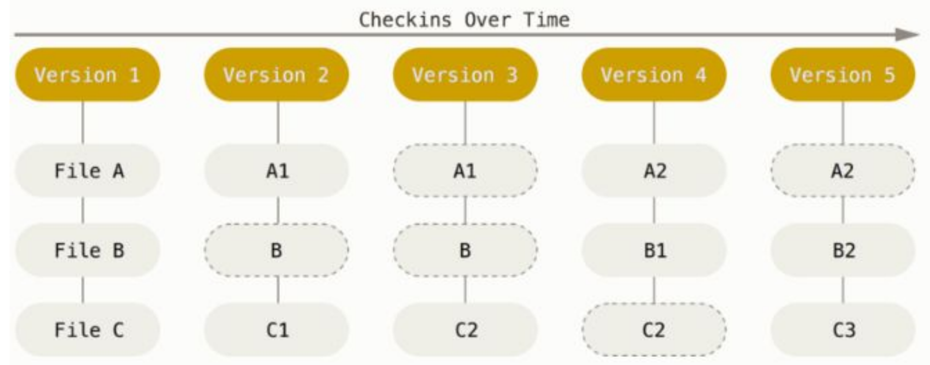
Untracked files are everything else — any files in your working directory that were not in your last snapshot and are not in your staging area.



How Git stores information

Every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.

To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots.



Branching

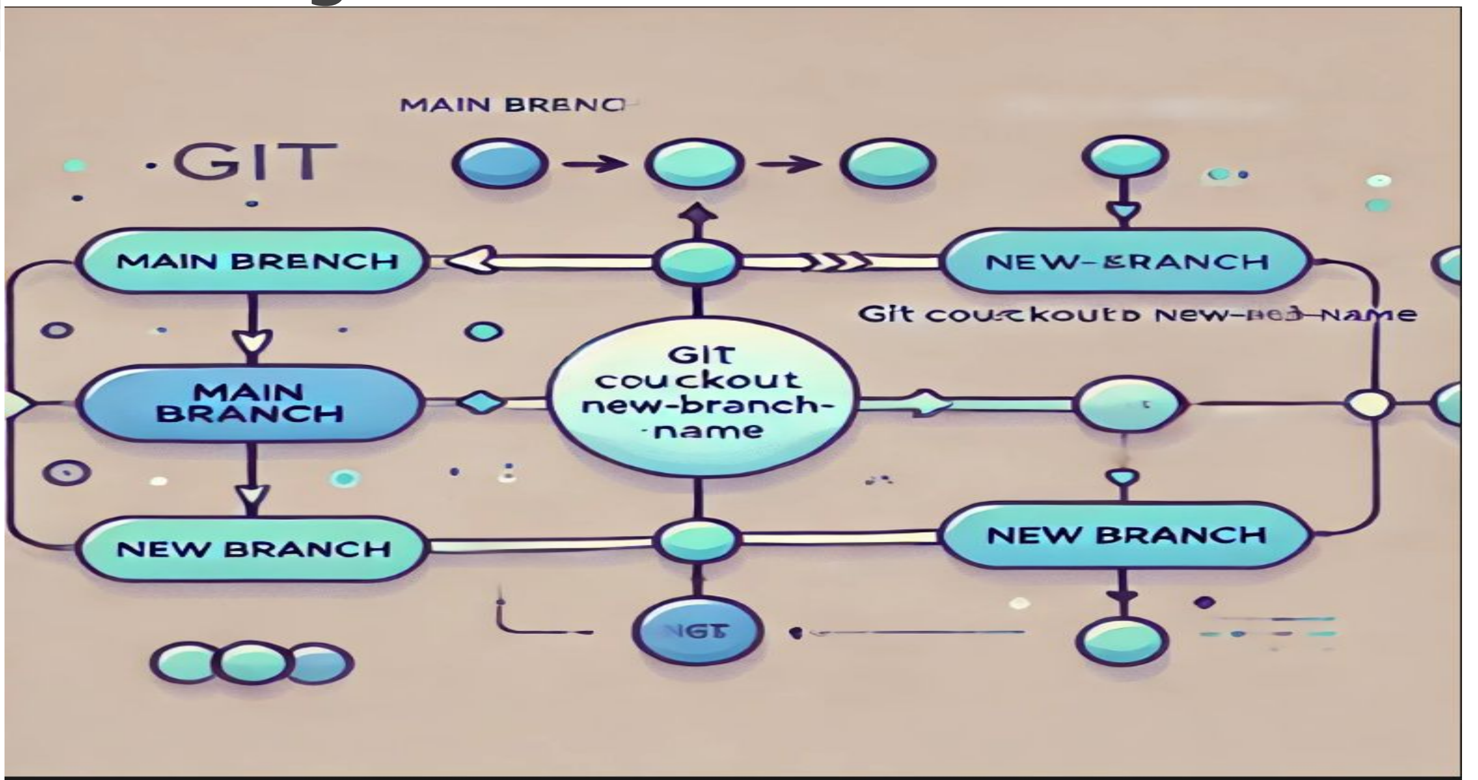
A branch in Git is **a simple movable pointer** to one of the commits.

By default, the name of the main branch in Git is **main (master)**.

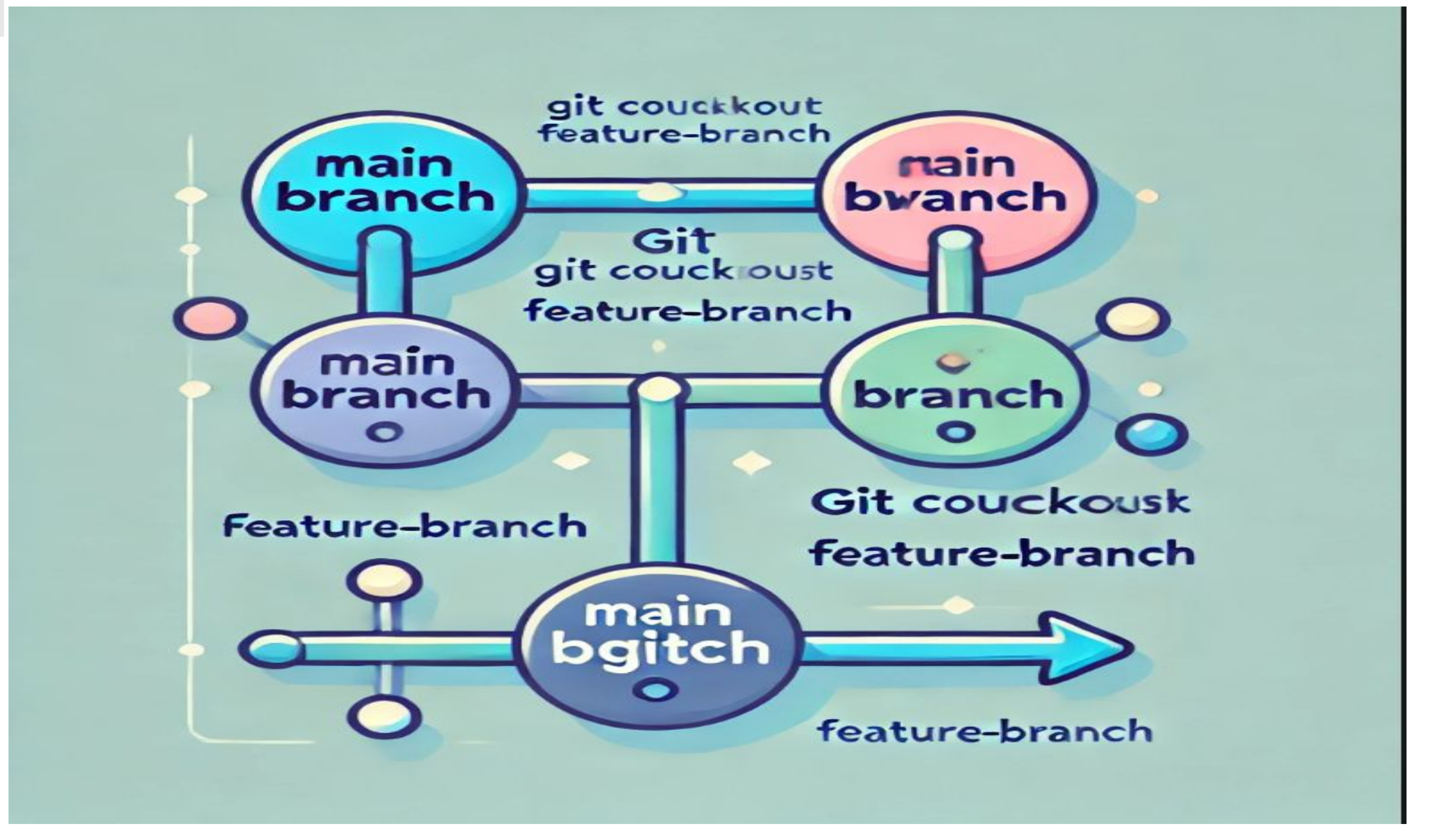
Once you start making commits, the **main** branch will always point to the **latest** commit. Each time you create a commit, the main branch pointer will **move** to the next commit automatically.

How does Git determine **which branch you are on**? It stores a special **HEAD** pointer.

Creating a new branch



Switching branches



Main Git commands

- git init - to initialize empty local repository
- git add <path> - to add files to index for the next commit
- git status - information about current repository satus
- git commit - to make a commit
- git remote add origin <URL> - to add a remote with “origin” name
- git log - to view commits
- git branch <name> - to create a branch with specified name
- git checkout <name> - to switch current branch to specified
- git config --global user.email <email> - to set up global email to authorize commits
- git config --global user.name <username>- to set up global username to authorize commits
- git push - to send changes to the remote repository

Best practices

- Always create “feature” branch when you start working on a new task.
- Before finishing the work, ***pull*** latest changes from the main branch and ***rebase*** main branch to keep your “feature” branch up-to-date.
- Create a ***pull request*** when you want to push your work to the main branch.
- ***squash/fixup*** meaningless commits in your branch
- Organise your commit messages
- Don’t commit generated files
- Make incremental, small changes
- Write descriptive commit messages

- ***NEVER push to the main branch directly!***

Questions?

SOLID

Single responsibility principle

A module should be responsible to one, and only one, actor. The term actor refers to a group (stakeholders, other code) that requires a change in the module.

Violation can result in: unmaintainable code, God object.

Open-closed principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Violation can result in: unintentional behaviors of existing components, fragile tests.

Liskov substitution principle

If *S* is a subtype of *T*, then objects of type *T* in a program may be replaced with objects of type *S* without altering any of the desirable properties of that program.

Violation can result in: different behaviour of components from same class hierarchy.

Interface segregation principle

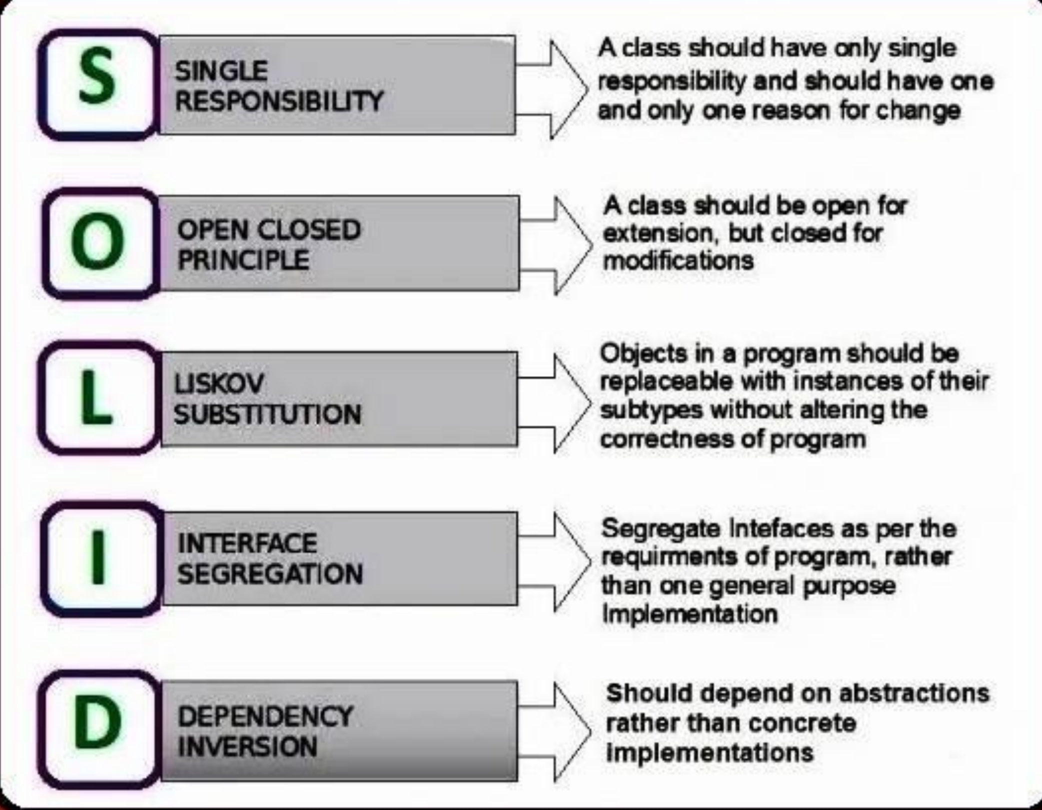
No code should be forced to depend on methods it does not use.

Violation can result in: God objects

Dependency inversion principle

- High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

Violation can result in: highly coupled code.

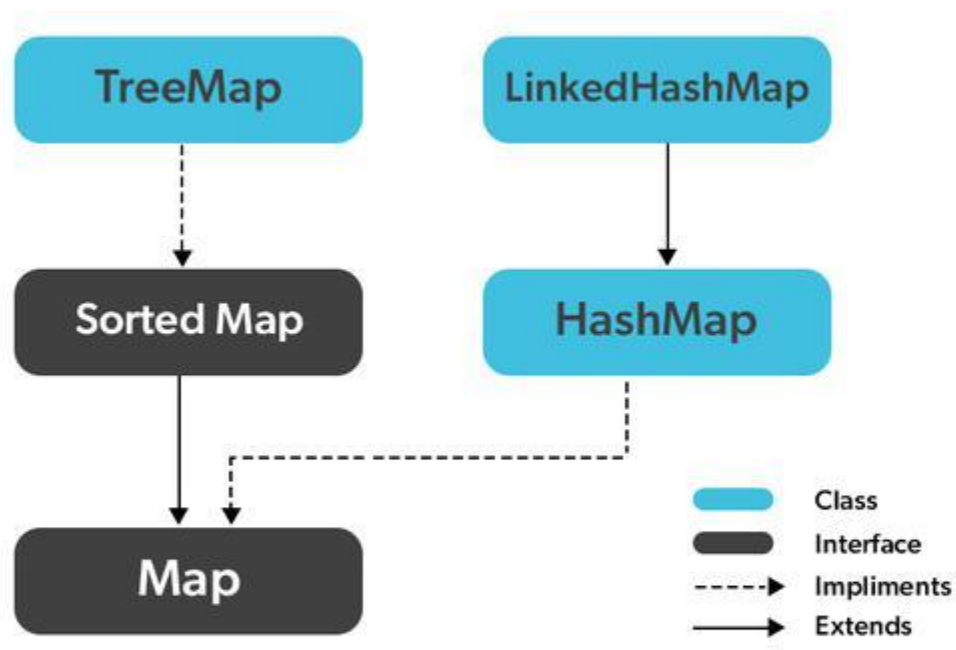

SOLID PRINCIPLES
S
SINGLE RESPONSIBILITY
A class should have only single responsibility and should have one and only one reason for change
O
OPEN CLOSED PRINCIPLE
A class should be open for extension, but closed for modifications
L
LISKOV SUBSTITUTION
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of program
I
INTERFACE SEGREGATION
Segregate Intefaces as per the requirments of program, rather than one general purpose implementation
D
DEPENDENCY INVERSION
Should depend on abstractions rather than concrete implementations

Questions?

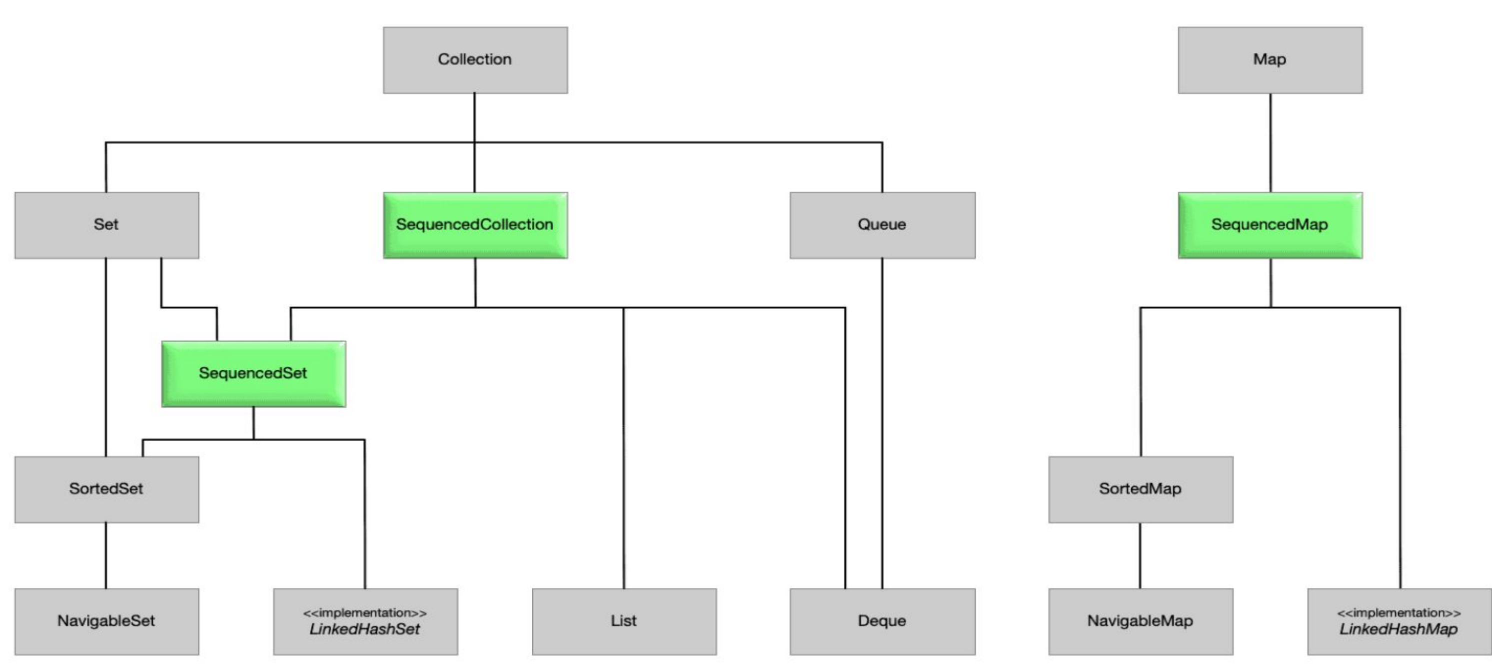
Collections

Collection hierarchy

Map hierarchy



Collection hierarchy from Java 21



List<E>

- Can contain duplicates
- Preserves order of insertions
- Allow access to elements by index
- Main implementations: ArrayList - based on array; LinkedList - based on nodes, which contain references to next and previous nodes

Queue<E>

- Access of elements order depends on concrete implementation
- Can not contain nulls
- The size can be constrained
- Deque (double-ended queue) support insertion, removal and access of elements from both ends.
- Main implementations: ArrayDeque - double-ended queue with unlimited size based on array; LinkedList; PriorityQueue- sorted queue with unlimited size.

Map<K,V>

- Operates on key-value pairs
- Heavily depends on “hashcode” and “equals” implementations of elements
- Keys are unique
- Each key is associated with only one value
- Main implementations: HashMap - based on the notion of Hashtable data structure; TreeMap - sorts pairs by key; LinkedHashMap - preserves insertion order.

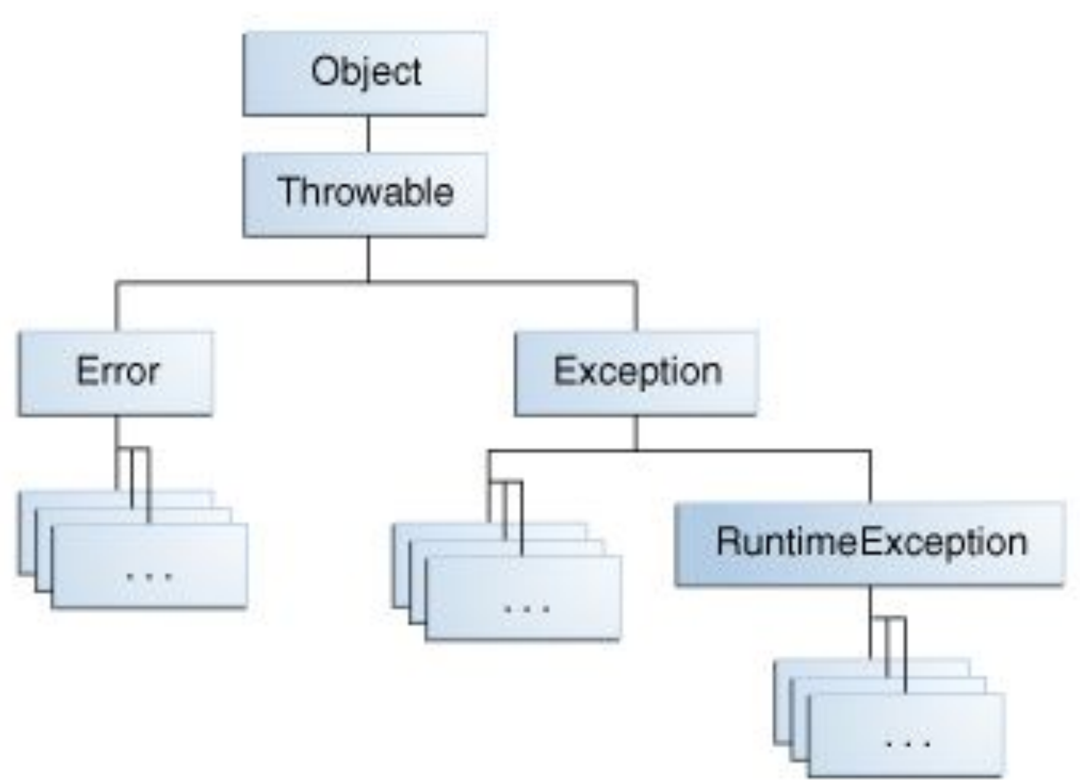
Set<E>

- Can not contain duplicates
- Main implementations: HashSet - based on HashMap; TreeSet - based on TreeMap

Questions?

Exceptions

Hierarchy



Error

Error – critical errors in the system. Not intended to be caught and dealt with (OutOfMemoryError, StackOverflowError).

Checked exceptions

All exception except RuntimeException, Error and their subclasses are checked. Method must declare in signature all possible checked exceptions, which can be thrown in its body.

void someMethod() throws FileNotFoundException { … }

`IOException`

`SQLException`

`FileNotFoundException`

`ClassNotFoundException`

Unchecked exceptions

RuntimeException, Error and their subclasses are unchecked. You can declare unchecked exception in method's signature, but it's not mandatory. Also it's not mandatory to catch them.

`NullPointerException`

`ArrayIndexOutOfBoundsException`

`ArithmeticException`

try-catch-finally

Allows to catch exception, thrown from try block, and resolve it. Finally block intended for code required to be invoked regardless.

```
try {
...
} catch (SomeException | SomeOtherException e) {
...
} finally {
...
}
```

Some rules

- Blocks catch and finally are optional. At least one of them should be present alongside with try
- Multiply exceptions can be resolved in a similar manner, if they are declared separated with a vertical bar
- There can be multiple catch blocks, but they need to be in order from the most specific to the most generic exception in hierarchy

try-with-resources

Allows to declare one or multiple variables (subtype of AutoClosable). “close” method is invoked by the end of “try-with-resources” execution

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {
        while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException  fnfe) {
        fnfe.printStackTrace();
}
```

Thank you for your attention!

Links

- Git book

Homework

- Pick a project
- Implement required functionality
- User should be able to interact with the application using console interface:
  apartment register
  order open
- You can omit tests for now
- You can store state in memory for now
- Think a little bit ahead about what can be changed in the future. Plan an architecture to support such changes.
- Any frameworks or libraries - forbidden!

Hotel

- Register a new apartment to manage given its price
- Reserve an apartment for a client given it’s currently free
- Release reservation for an apartment given it’s currently reserved
- List apartments (with pagination) sorted by ID, price, reservation status, client’s name

Bookstore

- Open a new order given total price and at least one book
- Cancel an order given it's currently opened
- Complete an order given it's currently opened
- List orders (with pagination) sorted by ID, total price, opening timestamp, closing timestamp, status

Car service

- Open a new order given its price
- Assign one or more repairers to the order given it’s currently opened
- Complete an order given it’s currently opened and has assigned repairers
- Cancel an order given it’s currently opened
- List orders (with pagination) sorted by ID, price, opening timestamp, completion timestamp, status