

Summary of CPEN 221

Tom Wang

Fall, 2023

1 Intro

Topics going to be covered in CPEN 221:

- Specifications
- typing and simple static analysis
- testing
- abstraction (procedural abstraction and data abstraction)
- abstract data types
- type hierarchies
- concurrency

2 Types

A type is a set of values and the operations that are permitted on the values.

2.1 OOP

In OOP, reference types are objects and primitive types are not. The name of the object is the reference to the object.

The object will have a constructor so that we can construct/create the object.

2.2 User-defined types

The intention of a type is that all interaction with values of that type be through the operations defined for that type.

Defensive programming:

Build a wall to limit the access from others. E.g., `private`.

2.3 Type safety

Type safety refers to a programming language's support for detecting type errors.

- Static analysis: Analysis performed during the compilation stage (or before a program is executed)
- Dynamic analysis: Runtime analysis

Languages that can support stronger static analyses typically lead to stronger correctness guarantees.

3 Exception

3.1 Definitions

Exceptions are contracts that are supported by multiple programming languages. Exception mechanisms allow programmers to indicate problems and mechanisms include techniques for method clients to recover from problematic situations in a reasonable manner.

Java provides **throw** mechanism to indicate a problem or special situation. **catch** mechanisms can handle an exception thrown by the method used.

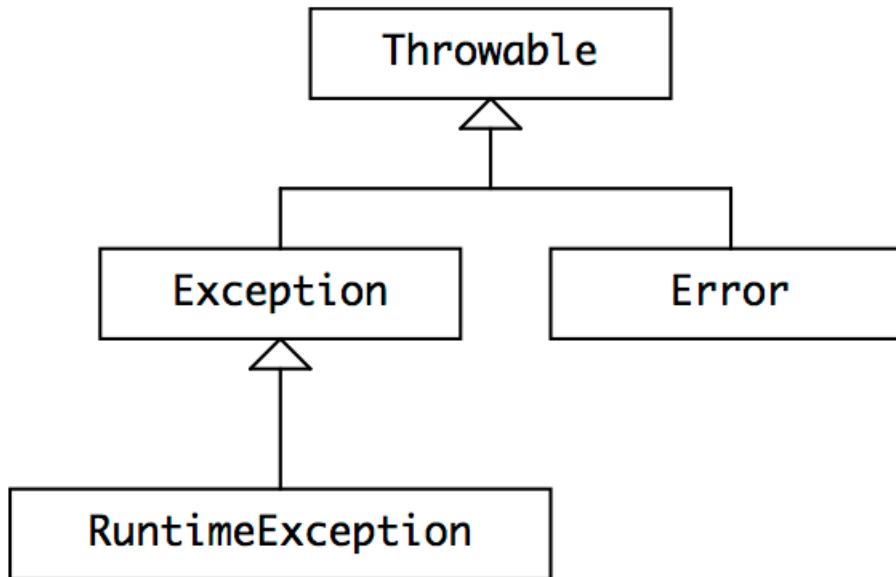
3.2 Type of exception

Java provides two exception types: **checked and unchecked**.

- Checked exception means it is checked by the compiler. It must be handled or declared the exception itself.
- Unchecked exceptions are used to signal bugs. These exceptions are not expected to be handled.

- All exceptions may have a message associated with them. If not provided in the constructor, the reference to the message string is null.

3.3 Throwable hierarchy



Throwable is the class of objects that can be thrown or caught. Any object used in a throw or catch statement, or declared in the throws clause of a method, must be a subclass of Throwable.

Error is a subclass of Throwable that is reserved for errors produced by the Java runtime system, such as `StackOverflowError` and `OutOfMemoryError`. For some reason, `AssertionError` also extends Error, even though it indicates a bug in user code, not in the runtime. Errors should be considered unrecoverable, and are generally not caught.

- `RuntimeException`, `Error`, and their subclasses are unchecked exceptions. The compiler doesn't require these to be declared in the throws clause of a method that throws them and doesn't require these to be caught or declared by a caller of such a method.
- All other throwables — `Throwable`, `Exception`, and all of their subclasses except for those of the `RuntimeException` and `Error` lineage — are checked exceptions. The compiler requires these exceptions to be caught or declared when they can be thrown.

When you define your own exceptions, you should either subclass `RuntimeException` (to make it an unchecked exception) or `Exception` (to make it checked).

3.4 Exception design considerations

- If you design a method to have its own (new) exception, you have to create a new class for the exception.
- If you call a method that can throw a checked exception, you have to wrap it in a try-catch statement (even if you know the exception will never be thrown).

Refined rule: You should use an unchecked exception only to signal an unexpected failure (typically, a bug), otherwise, you should use a checked exception.

4 Computing system

4.1 fetch-and-execute cycle

CPU fetches instruction from memory and then executes it.

Registers keep the values inside the CPU, program counter keeps track of the instruction address.

4.2 Asynchronous Events

CPU checks data over and over again and executes when the desired data arrives. It is called polling. However, it wastes performance to keep waiting.

Instead, the CPU gets interrupts, which stalls the current task and handles interrupts.

Processes are individual tasks that the CPU is working on. Note that processes can have multiple streams of execution called threads.

4.3 Java Virtual Machine

Lowlevel language can be translated directly to assembly and then machine language by the compilers. However highlevel languages like Java need an intermediate step.

JVM is an interpreter that translates instructions one by one. Compilers translate Java to Java Bytecode program, which is a machine language for JVM. Then Java interpreter translates Java Bytecode to the local machine language. So it can be used across platforms.

To solve the problem that it takes time to translate between JVM and local machine, Java introduced justintime compilers which translates from JVM to machine code while compiling to reduce time.

5 Program

5.1 Computation and Memory

In the old days, the compiler explicitly decides the address of data. However, the challenge is that no other program can use it anymore.

The modern approach is to indicate an offset of the base address. It allows multiple programs to run at the same time. Also, it allows the same data to be accessed by different programs giving different offsets.

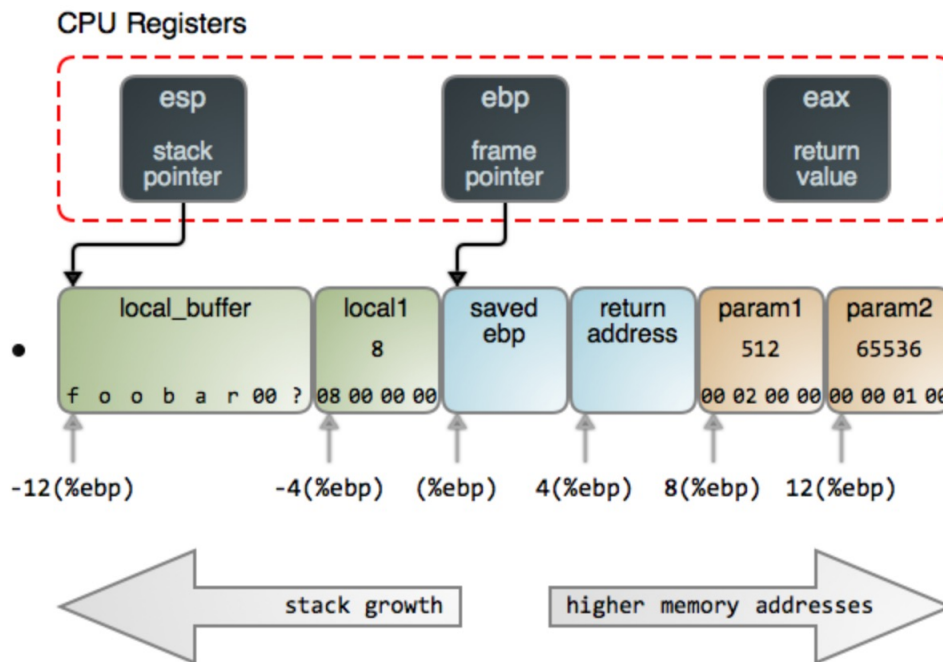
Compilers use offset from a stack pointer to track address. Stack is **Last-in-first-out**. So the last function called will be the first function executed.

Memory is therefore divided into a few regions, one of which is called the stack and this is where local variables declared in a function are stored. Another region of memory that is useful is called the heap, and data whose sizes cannot be known ahead of time (at compile time) are stored on the heap.

CPU contains **Program Counter** (PC) to keep track of the memory address of the instruction being executed. Normally, pc is incremented by 1 unless conditional statements and function calls are given.

5.2 Call Stacks and Stack Frames

When a function is called, a **stack frame** is created to support the function's execution. The stack frame contains the function's local variables and the arguments passed to the function by its caller. The frame also contains housekeeping information that allows the called function (the callee) to return to the caller safely.



The stack pointer, **esp**, always points to the last item pushed in, which is the top of the stack. It specifically points to the first byte occupied by *local buffer*, which is the lowest memory address in a stack containing live data.

Base pointer or frame pointer, **ebp**, points to a fixed location within the stack frame of the function currently running and provides a stable reference point for **accessing arguments and local variables**. (Using offset) Unlike *esp*, *ebp* is mostly maintained by program code with little CPU interference.

Finally, the **eax** register is used by convention to transfer return values back to the call for most C data types.

Check out CN5.3. For more details.

Summarize the steps:

1. “call main”:push return address onto the stack, jump into main
2. “push ebp”:save current ebp register values
3. “mov esp, ebp”:copy esp to ebp

These are function prologue. The current value of *ebp* is saved to the top of the stack. *esp* is copied to *ebp* to establish a new frame.

After some function calls it will call back to the original position.

6 Code testing and review

6.1 Software Validation

The purpose of validation is to uncover problems in a program and thereby increase one's confidence in the program's correctness.

Validation includes:

- Formal reasoning: usually called verification. Not so possible for large programs.
- Testing: Running the program on carefully selected inputs and checking the results.
- Code review: Having someone else carefully read your code and reason informally about it.

As a tester, you want to make the program fail.

We should test early and often. You write tests before you even write any code. The development of a single function proceeds in this order:

1. Specification for the function
2. Tests that exercise the function
3. Actual code. Once you pass the tests, you are done.

6.2 Why software testing is hard

Here are some approaches that do not work well:

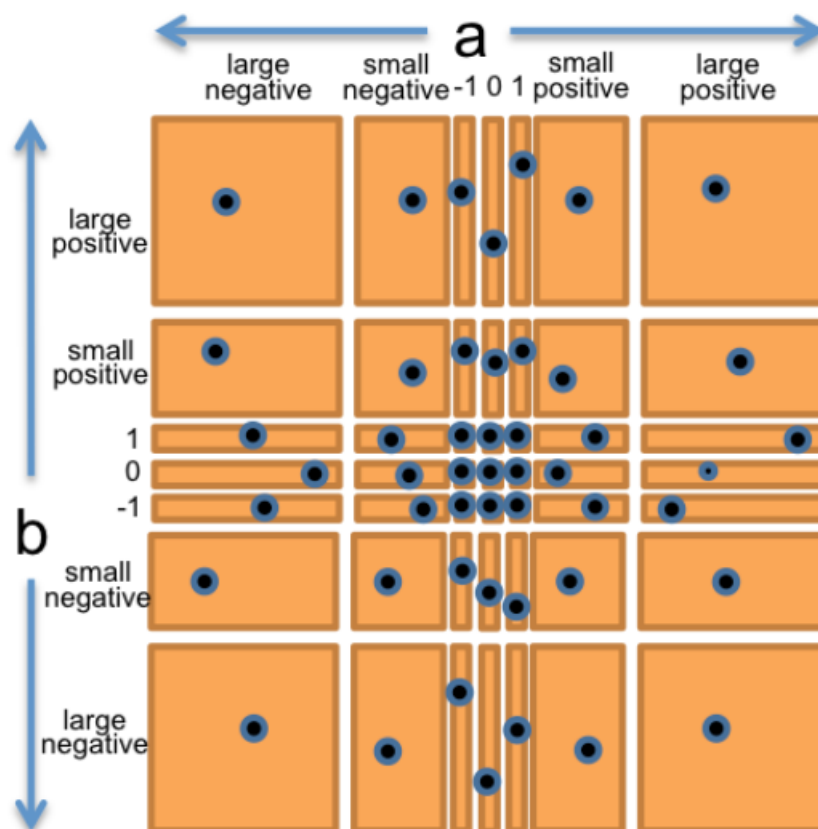
- Exhaustive testing: Space of possibility is too large to cover.
- Haphazard testing: (Just try it and see if it works) does not do anything. It will not increase confidence in the program.
- Random or statistical testing: Randomly testing a small portion of the software is not working. Software behaves discontinuously and discretely across the space of possible input.

6.3 Partitioning

Test suite: a set of test cases that cover different aspects of a program's behaviour.

We divide the input space into **subdomains**, each consisting of a set of inputs. Taken together the subdomains completely cover the input space and every input lies in at least one subdomain. Then we choose one test case from each subdomain, and that is our test suite.

Example: `BigInteger.multiply()`



Bugs often occur at the boundary between subdomains. There are a few reasons:

- Programmers often make off-by-one mistakes. For example, mess up $>$ and \geq .
- Some boundaries may need to be handled as special cases in the code.

- Boundaries may be places of discontinuity in the code's behavior. For example, `int` variable grows beyond its maximum positive value.

6.4 Two Extremes for Testing

After partitioning the input space, we can choose how exhaustive we want the test suite to be:

- Full Cartesian product: Every legal combination of the partition dimensions is covered by one test case.
- Cover each part: Every part of each dimension is covered by at least one test case, but not necessarily every combination.

6.5 Blackbox and Whitebox Testing

Specification is the description of the function's behaviour — The types of parameters, type of return value, and constraints and relationships between them.

Blackbox testing means choosing test cases only from the specification, not the implementation of the function.

Whitebox testing means choosing test cases with knowledge of how the function is implemented. You must take care that your test cases don't require specific implementation behaviour that isn't specifically called for by the spec.

6.6 Coverage

- Statement coverage: is every statement run by some test case?
- Branch coverage: For every if or while statement, are both the true or false directions taken by some test case?
- Path coverage: is every possible combination of branches — every path through the program — taken by some test case?

It is crucial to rerun your tests when you modify your code. It prevents the program from regressing — introducing other bugs. Running all your tests after every change is called regression testing.

6.7 Coding style recommendations

There are only two constants that many computer engineers and computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2. (Okay, three constants.)

Other constant numbers need to be explained. One way to explain them is with a comment, but a far better way is to declare the number as a constant with a good, explanatory name.

Do not reuse parameters and variables. One purpose for each variable.

Add “final” to parameters or variables so that it will never be reassigned.

Use good names so that it is self-descriptive.

Three Cs:

- Correct: find bugs
- Comprehensible: understandable
- Changeable: Readiness for change was considered by writing tests that only depend on behavior in the spec.

7 Specifications

A specification is a description of what an artifact does.

The contract between the producer and the consumer can be described using preconditions and postconditions.

If the consumer promises to use the artifact in a particular fashion (this is the precondition) then the producer guarantees that the artifact will behave in the intended manner (which is the postcondition)

Precondition: The requirements on the input values.

Postcondition: The description of the output values.

Frame condition: Explain which element might change during the execution, avoid unintended behavior.

7.1 The role of a spec

With a spec:

- the client (the user of a method or a class) agrees to rely only on information in the description/specification of the method or class for their part, and

- the implementer promises to support everything in the specification, and outside of the specification has perfect liberty to make implementation decisions.

A specification of a method can be concerned with the parameters and return value of the method, but it should never discuss the local variables of the method or private fields of the method's class.

7.2 Designing specifications

- Deterministic: Does the spec define only a single possible output for a given input or allow the implementor to choose from a set of legal outputs?
 - Deterministic: when presented with a state satisfying the precondition, the the outcome is determined.
 - Non-Deterministic code is code that you expect to sometimes behave one way and sometimes another.
- Declarative: Does the spec just characterize what the output should be or does it explicitly says how to compute the output.
 - Operational specs give a series of steps that the method performs; pseudocode descriptions are Operational
 - Declarative specs just give properties of the final outcome and how it is related to the initial state.
- Strong: Does the spec have a small set of legal implementations or a large set?

A specification A is stronger than or equal to a specification B if:

- A's precondition is weaker than or equal to B's. (More or equal amount of allowed input)
- A's postcondition is stronger than or equal to B's, for the states that satisfy B's precondition. (Know more about the return value and outcome)

8 Debugging

The best defense against bugs is to make them impossible by design.

Static checking: catch them at compile time.

Dynamic checking: The system throws errors automatically. E.g. `IndexOutOfBoundsException`.

Make sure you know the difference between variables and pointers. Making pointers final only keeps the address unchanged but not the thing it points to.

Fail fast to localize bugs. Use “*assertvariable*” to check for errors. Following are a few things you should assert:

- method argument requirements
- method return value requirements
- Covering all cases: if a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to block the illegal cases.

Many assertion mechanisms are designed so that assertions are executed only during testing and debugging and turned off when the program is released to users. So do not use it on a move directly. Make sure the program works without the assertion line.

8.1 Modularity and Encapsulation

Better software design:

- Modularity: Dividing up a system into components or modules. Each can be designed, implemented, tested, reasoned about and reused separately from the rest of the program.
- Encapsulation: Building walls around a module so that the module is responsible for its own internal behavior and bugs in other parts of the system cannot damage its integrity.
 - Access control: use ‘public’ and ‘private’ to control the visibility and accessibility of variables and methods.

- Variable scope: The scope of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable. Keeping variable scopes as small as possible makes it much easier to reason about where a bug might be in the program.

9 Mutability

An important idea in computing (and in reasoning about computation) is that of mutable and immutable values.

9.1 Pass parameters

Passing a mutable reference type could cause hazards! Be extra careful about it!

Defensive Copying: Always return a copy of the mutable type instead of directly. Same thing for passing them! Do not directly return the address of a pointer! That will allow other programs to access the data at that address.

Immutability can be more efficient than mutability because immutable types never need to be defensively copied.

9.2 Aliasing

Aliases: multiple references on the same thing.

Maybe we should just always indicate in the specs that we are returning a copy of a mutable type!

9.3 Iterator

Iterator is a mutable class that could be used to iterate through different data types. Methods in it would be able to mutate the desired passed in parameters properly.

10 Recursion

Recursion is a problem-solving process wherein to solve an instance of a problem one would need to solve other instances of the same problem.

We can define a recursive process (or a recursive object) as having two properties:

- **a set of base cases**, that do not rely on recursion and indicate terminal steps (these are also called termination conditions);
- **a recursive step**, that reduces an instance of the process to other instances of the same process, in the direction of one of the base cases.

10.1 Recursion and Induction

Let $P(n)$ be a statement for the n^{th} natural number. We want to show that this statement is true for all $n \geq 0$

The idea of induction:

- Show $P(0)$ is true. (Base case)
- Establish that if $P(1), \dots, P(m)$ were true, then $P(m + 1)$ is true.

10.2 Binary Search

Binary Search is a recursive approach to searching through a sorted array.

The intuition is that we can look at some element in the array, and if it is what we are looking for then we are done. If it is not, then we can decide which part of the array to search in based on whether the search value is smaller or larger than the value we have picked.

```

* @param intArray the array to search in,
*     should not be null,
*     and should be sorted in non-descending order
* @param x the value to search for
* @param startIndex the starting index in intArray
*     for our search, >= 0
* @param endIndex the final index in intArray
*     for our search, < intArray.length
* @return true if x exists in intArray and false otherwise
*/
public static boolean binarySearch(int[] intArray, int x, int startIndex, int endIndex) {

    // something missing here

    int midIndex = (startIndex+endIndex) / 2;
    if (intArray[mid] == x) {
        return true;
    }
    else {
        if (intArray[mid] > x) {
            return binarySearch(intArray, x, startIndex, midIndex - 1);
        }
        else {
            return binarySearch(intArray, x, midIndex + 1, endIndex);
        }
    }
}

```

Warning!!!!!! The implementation is missing a termination situation where the the desired value is not in the data: if (startIndex >= endIndex) returns false;

10.3 Memorization

We can cache the results to prevent repetitive executions.

- Use a hashmap to store the results calculated already.
- Then check if the map contains the key to see if the calculation was done before.
- If contained, directly return the calculated value.
- It will save a lot of time!

11 Abstract Data Type

A datatype is a collection of objects/items and the operations that you can perform on those objects. Which means it is independent of the representation. As long as you can do the exact defined job, you are it!

11.1 Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—public, or package-private (no explicit modifier).
- At the member level—public, private, protected, or package-private (no explicit modifier).

A class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as package-private), it is visible only within its own package (packages are named groups of related classes)

For members, there are two additional access modifiers: `private` and `protected`. The `private` modifier specifies that the member can only be accessed in its own class. The `protected` modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.

Access Levels

Modifier	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
<code>no modifier</code>	Y	Y	N	N
<code>private</code>	Y	N	N	N

11.2 What Abstraction Means

- Abstraction.

Omitting or hiding low-level details with a simpler, higher-level idea.

- Modularity.

Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.

- Encapsulation.

Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system cannot damage its integrity.

- Information hiding.

Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.

- Separation of concerns.

Making a feature (or “concern”) the responsibility of a single module, rather than spreading it across multiple modules.

The key idea of data abstraction is that a type is characterized by the operations you can perform on it

11.3 Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as mutable or immutable. Sometimes a type will be provided in two forms, a mutable and an immutable form. `StringBuilder`, for example, is a mutable version of `String`

The operations of an abstract type can be classified as follows:

- Creators create **new objects** of the type. A creator may take an object as an argument, but not an object of the type being constructed. In many programming languages, constructor methods are creators although there are other creator operations too.
- Producers create new objects from old objects of the type.
- Observers take objects of the abstract type and return objects of a different type.
- Mutators change objects. The `add` method of `List`, for example, mutates a list by adding an element to the end.

11.4 Designing an Abstract Type

It is better to have **a few, simple operations** that can be combined in powerful ways, rather than lots of complex operations.

Each operation should have a well-defined purpose and should have a coherent behavior rather than a panoply of special cases.

The set of operations should be adequate in the sense that there must be enough to do the kinds of computations clients are likely to want to do.

Should not mix generic and domain-specific features.

Critically, a good abstract data type should be **representation independent**. This means that the use of an abstract type is independent of its representation (the actual data structure or data fields used to implement it), so that changes in representation have no effect on code outside the abstract type itself.

11.5 Interfaces

An interface in Java is a list of method signatures, but no method bodies. A the class implements an interface if it declares the interface in its implements clause, and provides method bodies for all of the interface's methods.

One advantage of this approach is that the interface specifies the contract for the client and nothing more. The implementation is kept well and truly separated, in a different class altogether.

Another advantage is that multiple representations of the abstract data type can co-exist in the same program, as different classes implementing the interface.

- Defining an interface:

An interface can extend any number of interfaces. Example below:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3
{
    //interface body.
}
```

- Implementing an interface:

There is also a functional interface in Java. Basically, an interface that contains exactly just 1 abstract method.

11.6 Invariants

The property of a good abstract data type is that it preserves its own invariants. An invariant is a property of a program that is always true, for every possible runtime state of the program.

Representation exposure: code outside the class can modify the representation directly.

Immutable wrappers:

For example, `Collections.unmodifiableList()` takes a (mutable) `List` and wraps it with an object that looks like a `List`, but whose mutators are disabled `set()`, `add()`, `remove()` throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list), and get an immutable list.

Establishing invariants:

- makes the invariant true in the initial state of the object.
- ensures that all changes to the object keep the invariant true.
- Define this ADT by available operations on it.
 - creators and producers must establish the invariant for new object instances
 - mutators and observers must preserve the invariant.

Structural induction: if an invariant of an ADT obeys:

- established by creators and producers
- preserved by mutators and observers
- no representation exposure occurs

Then the invariant is true of all instances of the ADT.

12 RI and AF

The Abstraction Function maps a concrete representation to the abstract value it represents. The Rep Invariant specifies the legal values of the representation and should be checked at runtime with `checkRep()`.

12.1 Connecting specs and implementations

- Representation invariant: $\text{Object} \rightarrow \text{boolean}$.
 - Indicates whether a data structure is well-formed.
 - Only well-formed representations are meaningful.
 - Defines the set of valid values of the data structure.
- Abstraction function: $\text{Object} \rightarrow \text{abstract values}$
 - What the data structure means (as an abstract value)
 - How the data structure is to be interpreted.
 - How do you compute the inverse, abstract value *to* Object?

12.2 Rep invariant

Rep invariant is a condition or set of conditions that must be true for an instance of an abstract data type (ADT) to be considered a valid and consistent representation.

It defines the allowed states and values that an object of the ADT can have. It serves as a consistency check to ensure that the object remains in a valid state throughout its lifecycle.

- Every abstract value is mapped to by some rep value.
- Some abstract values are mapped to by more than one rep value. (because representation is not a tight encoding. there is more than one way to represent)
- Not all rep values are mapped. (If not satisfying the precondition, it may cause some trouble. So not mapping them)

Have a method to check for rep invariant:

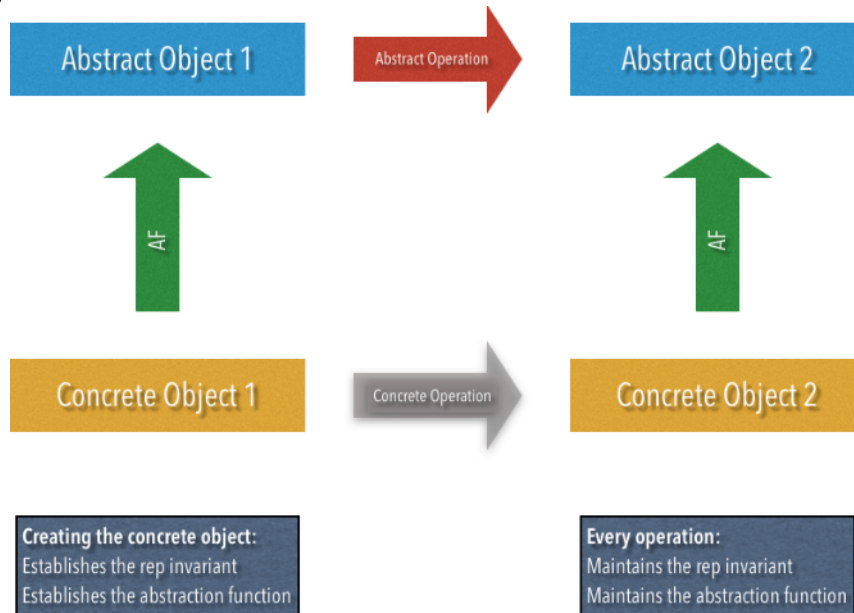
```
private void checkRep(){...}
```

Make sure to check for null.

12.3 Abstraction function

An AF defines a mapping between the internal representation of an ADT and the abstract, conceptual view of the data that the ADT is supposed to represent.

It helps bridge the gap between the internal representation and the external understanding of the data. It provides a way to interpret the internal state of an object in terms of its intended use.



12.4 Relationship between the two

- Rep invariant ensures that the internal state of an object remains consistent and valid.
- The abstract function provides a way to view the object in terms of its intended abstraction, allowing clients to interact with the object based on its conceptual model rather than its internal details.

12.5 How to write?

```
/**
 * This class represents the mathematical concept of a line segment.
 *
 * Abstract Invariant:
 * A line's start-point must be different from its end-point.
 */
public class Line {

    /** The x-coordinate of the line's starting point. */
    private int startX;

    /** The y-coordinate of the line's starting point. */
    private int startY;

    /** The x-coordinate of the line's ending point. */
    private int endX;

    /** The y-coordinate of the line's ending point. */
    private int endY;

    // Representation Invariant:
    // ! (startX == endX && startY == endY)
    //
    // Abstraction Function:
    // AF(r) = a line, l, such that
    //   l.start-point = (r.startX, r.startY)
    //   l.end-point = (r.endX, r.endY)

    /**
     * // requires p != null && ! p.equals(start-point)
     * // modifies this
     * // effects Sets end-point to p
     */
    public void setEndPoint(Point p) {
        ...
    }
}
```

13 Equality

13.1 Three ways to regard equality

1. Using an abstraction function: recall $f : R \rightarrow A$ maps concrete instances of a data type to their corresponding abstract values.

We say that a equals b if and only if $f(a) = f(b)$.

2. Using a relation: An equivalence is a relation $E \subseteq T \times T$ that is:

- reflexive: $E(t, t) \forall t \in T$
- symmetric: $E(t, u) \implies E(u, t)$
- transitive: $E(t, u) \wedge E(u, v) \implies E(t, v)$

To use E as a definition for equality, we would say that a equals b if and only if $E(a, b)$

Not quite know what this is.

3. How an outsider (a client) can observe about them: Observation means calling operations on the objects. So two objects are equal if and only if they cannot be distinguished by calling any operations of the abstract data type.

13.2 Comparing in programming

`==` operator compares references. It tests referential equality. So only true if the value is the same.

`equals()` compares object contents. The equals operation has to be defined appropriately for every abstract data type.

13.3 The object contract

The spec of **Object** class is so important that it is often referred to as the **Object Contract**. When you override the equals method, you must adhere to its general contract:

- equals must define an equivalence relation. (reflexive, symmetric, transitive)
- equals must be consistent.

- for a non-null reference, it should return false compared to null.
- hashCode must produce the same result for two objects that should return true.

Always override hashCode when you override equals!!!!!!!!!!

13.4 Equality of Mutable/Immutable Types

For immutable types:

- equals() should compare abstract values. This is the same as saying equals() should provide behavioral equality.
- hashCode() should map the abstract value to an integer.

So immutable types must override both equals() and hashCode().

For mutable types:

- equals() should compare references, just like ==.(behavioral equality)
- hashCode should map the reference into an integer.

So mutable types should not override equals() and hashCode() at all!

13.5 Autoboxing

Java may covert some primitive types to object types and back automatically.

Might not always work, although we see int and Integer equivalents usually.

13.6 CompareTo

In Java, a datatype that permits the ordering of instances can be implemented using a class that implements the interface Comparable. To implement the interface, one would need to implement a method compareTo.

compareTo should be consistent with equals: if equals(o) returns true then compareTo(o) should return 0.

13.7 Overloading and Overriding

Overloading: two unrelated methods with the same name. The compiler chooses which method to invoke based on compile-time types of all arguments (including the receiver)

Overriding: If the same name and parameter types, then multiple implementations of the same method family. The runtime system chooses which implementation to run based on the runtime type of the receiver (only).

14 Subtyping

14.1 functional interface and delegation

Functional interface: an interface that has exactly one method.

Lambdas: anonymous functions which are passed as parameters to other functions.

...

14.2 Subclassing

“Extends” key is used to define subclass relationships.

Polymorphism: Dynamic method invocation is referred to the runtime type although we might refer to the static type in programming.

If class B extends class A, it inherits the methods from A. B may choose to override those methods or rely on the implementation in A.

B is a subtype of A means every B object is also an A object.

A class can extend only one parent class but can implement multiple interfaces.

Liskov Substitution Principle(LSP): Subtypes must be substitutable for their supertypes. In particular, a subtype must fulfill the same contract as its supertype, so that clients designed to work with the supertype can use the subtype objects safely.

B is only a subtype of A if B’s specification is at least as strong as A’s specification.

14.3 Some wisdom

Inherited superclass methods can break the subclass’s rep invariant.

When a subclass overrides superclass methods, it may depend on how the superclass uses its own methods.

When a class is subclassed, it must freeze its implementation forever, or all its subclasses must evolve with its implementation. (If you change the superclass again, change all subclasses.)

In general, mutable counterparts of immutable classes should not be declared as subtypes.

14.4 Java Subtyping

First of all, need to clarify that “Java Subtyping” is not the same as the concept of subtyping.

Java types are defined by classes, interfaces, and primitives.

Java subtyping: either **B extends A** and **B implements A** declarations.

In a Java subtype, each corresponding method has:

- Same argument types. (If different, overloading: unrelated methods)
- Compatible (covariant) return types
- No additional declared exceptions

Java subtyping guarantees that: A variable’s run-time type (= the class of its run-time value) is a Java subtype of its declared type

```
Object o = new Date(); // OK
Date d = new Object(); // compile-time error
```

If a variable of declared (compile-time) type T holds a reference to an object of actual (runtime) type T’, then T’ must be a (Java) subtype of T

Corollaries:

- Objects always have implementations of the methods specified by their declared type
- If all subtypes are true subtypes, then all objects meet the specification of their declared type

14.5 Interfaces and Abstract Classes

An abstract class is a class that can only be subclassed, it cannot be instantiated

- Abstract classes can provide implementations for some instance methods, while interfaces cannot.
- To implement the type defined by abstract class A, class B must be a subclass of abstract class A (declared with extends). But any class that defines all of the required methods and follows the specification of the interface I can be a subtype of I (declared with implements).

15 Recursive Types

Recursive datatypes are those where an object of a type is defined in terms of the “smaller object” of the same type. Lists and trees can be an example of recursive datatypes.

Generally, it can be described as:

- Parent node holds child nodes.
- Child nodes can be representations of different nodes that all work as a recursive node. You can define as many types of nodes as possible and use methods and fields to rule each one’s behavior. Usually, using all types of nodes to extend a Node type could save our lives.
- Make sure there is an empty node meaning that the tree terminates here.

15.1 Null V.S. Empty

Null might seem to be an easier way of expressing Empty, however, using an actual object to represent Empty can allow us to call some functions that could be more informative and are less likely to cause an error.

Keep nulls out of your data structures, and your life will be happier.

15.2 Declared Type V.S. Actual Type

Note that type checking occurs at **compile time and run time**.

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

Here during the compile time, it is “List” and during run time it is “ArrayList”. This is why an interface can be used to declare a type but cannot be used to new it.

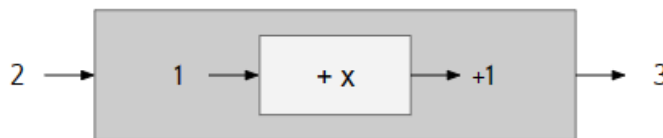
16 Lambdas and Streams

16.1 Lambda Calculus

Lambdas Calculus is an expression generated by the following grammar which can denote a function definition, function application, variable, or parenthesized expression:

$$expr \rightarrow \lambda var.expr \mid expr \quad expr \mid var \mid (expr)$$

For example: $\lambda x. (+ x 1)2$ represents the application of a function abstraction with a formal variable x and a body $+ x 1$ to the argument 2.



In java, we represent it as $x \rightarrow x + 1$

16.2 Java Lambdas

A Java Lambda is basically a **method without a declaration**, usually written as

$$\{\text{parameters}\} \rightarrow \{\text{body}\}$$

A lambda can have zero or more parameters separated by commas, and their type can be explicitly declared or inferred from the context. **Parenthesis are not needed** around a single parameter. $()$ is used to denote zero parameters. The body can contain zero or more statements. Braces are not needed around a single-statement body.

16.3 Streams

First, look at an example of using streams.

```
int sum(int[] arr) {
    return Arrays.stream(arr).reduce(0, (x,y) -> x+y);
}
```

$(x, y) \rightarrow x+y$ is a lambda.

```
int sum(int[] arr) {
    int sum = 0;
    for (int x : arr)
        sum += x;
    return sum;
}
```

A Stream in Java is a wrapper around a collection that keeps the collection immutable and also bundles an iterator for performing computation over the entire collection.

A more detailed example:

```
List<Transaction> groceryTransactions = new ArrayList<>();
for(Transaction t: transactions){
    if(t.getType() == Transaction.GROCERY){
        groceryTransactions.add(t);
    }
}
Collections.sort(groceryTransactions, new Comparator(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
    transactionIds.add(t.getId());
}
```

We can achieve the same objective using streams. And the resulting code is concise and expresses intent clearly.

```
List<Integer> transactionIds =
    transactions.stream()
        .filter(t -> t.getType() == Transaction.GROCERY)
        .sorted(comparing(Transaction::getValue).reversed())
        .map(Transaction::getId)
        .collect(toList());
```

The first step is to create a Stream of Transaction objects using stream() method on the list. We are then able to apply operations to this stream that do not mutate the original list. Instead, these operations return new streams.

There are a few ideas regarding streams that are good to indicate upfront:

- Sequence of elements: A stream provides an interface to a sequenced set of values of a specific element type. However, streams do not actually store elements; they are computed on demand.
- Source: Streams consume from a data-providing source such as collections, arrays, or I/O resources.

- Aggregate operations: Streams support SQL-like operations and common operations from functional programming languages, such as filter, map, reduce, find, match, sorted, and so on.

Furthermore, stream operations have two fundamental characteristics that make them very different from collection operations.

- Pipelining: Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as laziness and short-circuiting, which we explore later.
- Internal iteration: In contrast to collections, which are iterated explicitly (external iteration), stream operations do the iteration behind the scenes for you.

Lastly, we may be able to perform certain sub-operations in parallel if there is no dependency as regards the sequence of those operations. To do so in Java, we would create a parallel stream using the `parallelStream()` method.

16.3.1 Sequence: map, filter and reduce

Any datatype that has an iterator can qualify as a sequence. We will have three operations for sequences: map, filter and reduce.

- map **applies a unary function** to each element and returns a new list containing the results, in the same order.

$$\text{map} : (E \rightarrow F) \times \text{Seq} \langle E \rangle \rightarrow \text{Seq} \langle F \rangle$$

To express it directly and visually, define it using Python:

```
def map(f, seq): result = []
    for x in seq: result.append(f(x))
    return result
```

Java can do the same thing with a longer syntax. The crucial piece is how we pass the function we want to apply to each element of a stream when we invoke the `map()` operation.

Suppose we wanted to compute the square root of all elements in a list of non-negative integers, we could do the following in Java:

```
List<Double> sqrtList = Arrays.asList(1, 4, 9, 16)
    .stream() .map(Math::sqrt)
    .collect(toList());
```

Note that we are passing functions as values. The map function takes a reference to a function as its first argument, not the result of that function.

- filter is used to **select elements** from a stream based on a specified condition. It returns a new stream containing only the elements that satisfy the condition. Not that it does not modify its input list.

Condition is defined by a predicate, which is a function that returns a boolean value indicating whether an element should be included in the new stream.

$$\text{filter}: (E \rightarrow \text{boolean}) \times \text{Seq} \langle E \rangle \rightarrow \text{Seq} \langle E \rangle$$

If we define a filter in Python:

```
def filter(f, seq):
    result = []
    for x in seq:
        if f(x):
            result.append(x)
    return result
```

- reduce is used to **aggregate** the elements of a stream into a single value. It applies a **binary** operator to elements in the stream, accumulating the result as it processes the elements.

So one should define two lambda variables in the lambda function and apply a binary operator on them.

$$\text{reduce}: (F \times E \rightarrow F) \times \text{Seq} \langle E \rangle \times F \rightarrow F$$

reduce(f, list, init) combines the elements of the list from left to right, by starting with init=f(init, list[i]) $\forall 0 \leq i < \text{list.size}()$ in ascending order.

If you omit the `init`, then it will throw an exception if the list is empty otherwise choose `list[0]` as `init` and start from `list[1]`.

If you really want to do it in a different order, we can use `fold-right(f, list, int)`. It is exactly the same as reduction other than it starts from the end to the start.

Note that the return type does not necessarily have to be the same.

16.4 Parallel and Serial Streams

Streams are serial unless declared parallel.

Java runtime partitions a parallel stream into substreams. JVM handles the process of dividing the workload among threads, executing them in parallel, and then combining the results.

Since the partition is quite random, the behavior might be different all the time and execution time are not uniform across the multiple partitions.

It could be seen as a *concurrent* implementation of streams. So it would carry all the problems that concurrent programs have.

16.5 FirstClass Functions in Java

In Java, the only first-class values are primitive values and object references. But objects can carry functions with them in the form of methods.

For example, the “`Comparator< T >`” object is a form of using an object to pass functions.

17 Grammar and Regular Expressions

17.1 Grammars

A grammar defines a set of sentences, where each sentence is a sequence of symbols called *tokens/terminals*. A grammar is a set of productions, where each production defines a *nonterminal*.

A grammar consists of the following components:

- Terminals: The basic symbols of the language.
- Nonterminals: Symbols that can be replaced by a sequence of terminals and nonterminals.

- **Productions:** Rules that define how nonterminals can be replaced by sequences of terminals and nonterminals.
- **Start symbol:** The nonterminal from which the derivation of the language starts.

17.2 Regular Expressions

Regular expressions are a powerful tool for pattern matching and text manipulation. They are used to describe patterns in strings and can be used in various programming languages and tools.

Here are some commonly used regular expression constructs:

- **Character classes:** Matches any single character from a set of characters. For example, the regular expression `[abc]` matches either 'a', 'b', or 'c'.
- **Quantifiers:** Specify the number of occurrences of a pattern. For example, the regular expression `a{2,4}` matches 'a' repeated 2 to 4 times.
- **Anchors:** Specify the position of a pattern in a string. For example, the regular expression `^abc` matches 'abc' at the beginning of a string.
- **Grouping and capturing:** Allows grouping of patterns and capturing of matched substrings. For example, the regular expression `(ab)+` matches 'ab' repeated one or more times.
- **Alternation:** Matches one of several alternatives. For example, the regular expression `a|b` matches either 'a' or 'b'.
- **Escape sequences:** Used to match special characters. For example, the regular expression `\d` matches any digit.

Here are some examples of regular expressions:

- `\d{3}-\d{3}-\d{4}`: Matches a phone number in the format XXX-XXX-XXXX.
- `^[A-Za-z]+$`: Matches a string consisting of only letters.
- `^\d{4}$`: Matches a four-digit number.

- `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`: Matches an email address.

Regular expressions provide a concise and flexible way to describe patterns in strings. They are widely used in text processing, data validation, and search operations.

17.3 A parsing example

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ParameterParser {
    public static void main(String[] args) {
        String input = "name1:10,name2:3.14,name3:5";

        // Define the regular expression pattern
        String pattern = "(\\w+):\\s*(\\d+\\.\\d+|\\d+)";

        // Create a Pattern object
        Pattern regex = Pattern.compile(pattern);

        // Create a Matcher object
        Matcher matcher = regex.matcher(input);

        // Iterate through the matches
        while (matcher.find()) {
            String name = matcher.group(1);
            String value = matcher.group(2);

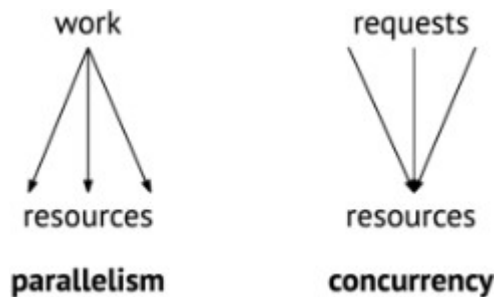
            System.out.println("Parameter name:" + name);
            System.out.println("Parameter value:" + value);
        }
    }
}
```

18 Intro to Concurrency and Parallelism

18.1 Process vs. Thread

- Process is an executing program. It contains its own memory space, file handles and other system resources allocated by the OS.
- One or more threads run in the context of one or more processes. Threads within the same process share the same memory space and resources allocated to that process.

18.2 Parallelism V.S. Concurrency



Parallel programming is about using additional computational resources to produce an answer faster.

Concurrent programming is about correctly and efficiently controlling access by multiple threads to shared resources.

18.3 Basic Threads and Shared Memory

Different threads can communicate by writing and reading fields of the same objects, in other words: share memory.

There may be more threads than processors which means not all threads will be running at the same time. **Scheduling** decides which thread uses the core at each time. It is not under programmers' control.

A thread may be waiting for something to happen before it continues. We use `join` to indicate.

Each thread has its own call stack and program counter, but all the threads share one collection of static fields and objects.

- When a new thread starts running, it will have its own new call stack. It will have one frame on it, which is like that thread's main, but not the actual main.
- When a thread returns from its first method, it terminates.
- Each thread has its own program counter and local variables, so there is no "interference" from other threads for these things.
- But! Static fields and objects will be unpredictable since you do not know the order of each thread accessing them.

18.4 Methods to communicate

- Shared-memory communication relies on **Aliasing**. Aliasing is when two different variables refer to the same object. Aliasing can be intentional or accidental.
- Message-passing: Threads do not share objects, instead, they send messages which is a copy of some data.
- Data flow: The programmer uses primitives to create a directed acyclic graph (DAG). A node in the graph performs some computation using inputs that arrive on its incoming edges.
- Data parallelism: does not have explicit threads or nodes running different parts of the program at different times. Instead, it has primitives for parallelism that involve applying the same operations to different pieces of data at the same time. For example, break down a large number operation into separate small numbers.

19 Fork-join Parallelism

Normal Threads procedure:

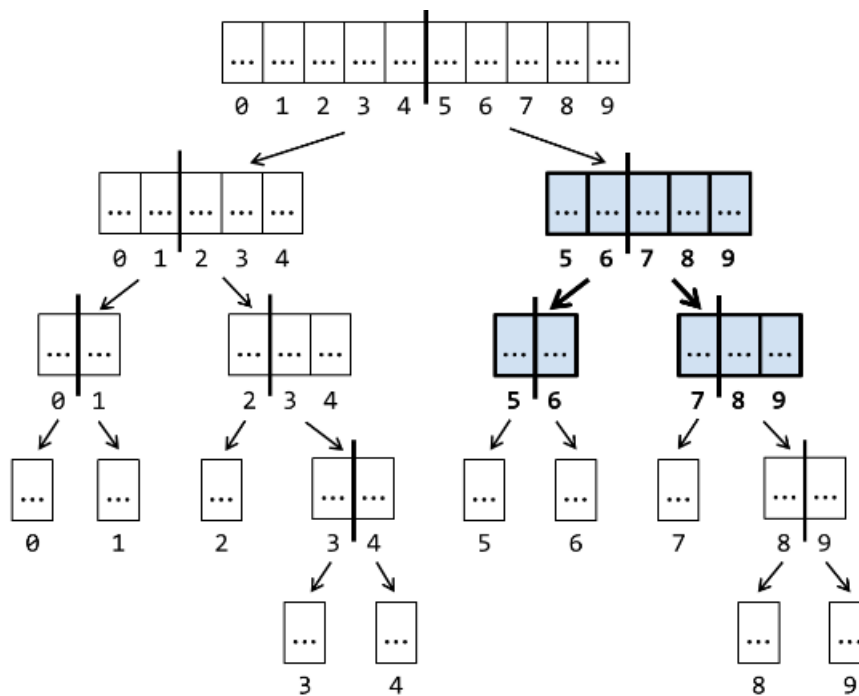
1. Define a new thread class that either **extends** `java.lang.Thread` or **implement** the thread interface, 'Runnable'. (Make sure to override the `run()` program since that is the key implementation)

2. Split the task roughly to the equal size that corresponds to the number of threads you want to have. (It is not possible to be perfect)
3. For each partition, new the specific thread and start it. (Might be a good idea to put threads into an array to be easily looked after.)
4. For each thread, wait for it to join, then collect the result. (This step is actually sequential. So while theoretically slicing the partition to be infinitely small can make sure work is balanced, adding up the result will take infinitely long since its sequential nature.)

19.1 Divide and Conquer Parallelism

The key idea is to:

1. If the range contains only one element, return that element as the sum. Else in parallel:
2. Recursively divide the task in half to left and right
3. Collect the results from nodes



```

class SumThread extends java.lang.Thread {
    static int SEQUENTIAL_CUTOFF = 1000;
    int lo; // fields for communicating inputs
    int hi;
    int[] arr;
    int ans = 0; // for communicating result
    SumThread (int[] a, int l, int h) { arr=a; lo=l; hi=h; }

    public void run() {
        if(hi - lo < SEQUENTIAL_CUTOFF) {
            for(int i=lo; i < hi; i++)
                ans += arr[i];
        } else {
            SumThread left = new SumThread(arr,lo,(hi+lo)/2);
            SumThread right= new SumThread(arr,(hi+lo)/2,hi);
            left.start();
            right.run();
            left.join();
            ans = left.ans + right.ans;
        }
    }

    public static int sum(int[] arr) {
        SumThread t = new SumThread(arr, 0, arr.length);
        t.run();
        return t.ans;
    }
}

```

Notice how the thread creates helper threads till the end. The normal calculation should not really only terminate at exactly 1 object. You can customarily design your own cutoff point as the base case.

A better improvement is that instead of starting the second thread at the right, you just directly run it. Creating a new thread at the right would result in an idle main thread waiting for joining, doing nothing!

```

SumThread left = new SumThread(arr, lo, (hi+lo)/2);
SumThread right = new SumThread(arr, (hi+lo)/2, hi);
left.start();
right.run();
left.join();
ans = left.ans + right.ans;

```

This way you do not waste threads since threads will be waiting for the two nodes to return values. Instead of letting it wait, we would rather let it do work directly.

Do notice that it takes time to create threads and work with threads. It might not be a good idea to do things that would not get too much help from threading.

19.2 ForkJoin Framework

```

import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

class SumArray extends RecursiveAction {
    static int SEQUENTIAL_THRESHOLD = 1000;

    int lo;
    int hi;
    int[] arr;
    int ans = 0;
    SumArray(int[] a, int l, int h) { lo=l; hi=h; arr=a; }

    protected void compute() {
        if (hi - lo <= SEQUENTIAL_THRESHOLD) {
            for (int i=lo; i < hi; ++i)
                ans += arr[i];
        } else {
            SumArray left = new SumArray(arr, lo, (hi+lo)/2);
            SumArray right = new SumArray(arr, (hi+lo)/2, hi);
            left.fork();
            right.compute();
            left.join();
            ans = left.ans + right.ans;
        }
    }
}

```

```

    }
}
}
class Main {
    static int sumArray(int[] array) {
        SumArray t = new SumArray(array, 0, array.length);
        ForkJoinPool.commonPool().invoke(t);
        return t.ans;
    }
}

```

Notice that the idea is similar, but we use different methods to achieve it.

- The entire program should have exactly one `ForkJoinPool`, so it makes sense to store it in a static field and use it for all the parallel algorithms in your program. The library's `commonPool` static method returns such an object for you.
- Note that inside the thread class, you use **fork**, **compute**, and **join** just like how we used **start**, **run** and **join** previously.
- But we use **invoke of the ForkJoinPool** outside of the class.

20 Network Programming

20.1 Client-Server Design Pattern

We have two processes:

- clients:
- servers:

Procedure:

1. A client initiates the communication by connecting to a server.
2. Then the client sends requests to the server, and the server sends replies back.
3. Finally, the client disconnects.

A server might handle connections from many clients concurrently and clients might also connect to multiple servers.

20.2 Network Sockets

20.2.1 IP address

A network interface is identified by an IP address. IPv4 address is a 32-bit number with four 8-bit parts. For example:

- 137.82.130.49 is the IP address of a UBC web server. Every address whose first octet is 137 is on the UBC network.
- 173.194.123.40 is the address of a Google web server.
- 127.0.0.1 is the loopback or localhost address: it always refers to the local machine. Technically, any address whose first octet is 127 is a loopback address, but 127.0.0.1 is standard.

20.2.2 Hostnames

Host names are names that can be translated into IP addresses. A single hostname can map to different IP addresses at different times, and multiple hostnames can map to the same IP address. For example

- `www.ubc.ca` is the name of UBC's web server. You can translate this name to an IP address yourself using `dig`, `host`, or `nslookup` on the command line, e.g.:

```
$ dig +short www.ubc.ca
137.82.130.49
```

- `localhost` is a name for 127.0.0.1. When you want to talk to a server running on your own machine, talk to `localhost`.

Translation from hostnames to IP addresses is the job of the Domain Name System (DNS).

20.2.3 Port Numbers

A single machine might have multiple server applications that clients wish to connect to, so we need a way to direct traffic on the same network interface to different processes.

Network interfaces have multiple ports identified by a 16-bit number from 0 (which is reserved, so we effectively start at 1) to 65535.

A server process binds to a particular port — it is now **listening** on that port. Clients have to know which port number the server is listening on.

There are some reserved ports, for example:

- Port 22 is the standard SSH port.
- Port 25 is the standard email server port.
- Port 80 is the standard web server port. When you connect to the URL `http://www.ubc.ca` in your web browser, it connects to 137.82.130.49 on port 80.

When the port is not a standard port, it is specified as part of the address. For example, the URL `http://130.2.39.10:9000` refers to port 9000 on the machine at 130.2.39.10.

When a client connects to a server, that outgoing connection also uses a port number on the client's network interface, usually chosen at random from the available non-well-known ports.

20.2.4 Sockets

A socket represents one end of the connection between the client and server.

- A **listening socket** is used by a server process to wait for connections from remote clients. In Java, use `ServerSocket` to make a listening socket, and use its `accept` method to listen to it.
- A **connected socket** can send and receive messages to and from the process on the other end of the connection. It is identified by both the local IP address and port number plus the remote address and port, which allows a server to differentiate between concurrent connections from different IPs, or from the same IP on different remote ports.

In Java, clients use a `Socket` constructor to establish a socket connection to a server. Servers obtain a connected socket as a `Socket` object returned from `ServerSocket.accept`.

20.3 Buffers

The size of data communication is unknown and not stable. So we just put them into buffers when data arrives.

The data going into or coming out of a socket is a stream of bytes.

In Java, `InputStream` objects represent sources of data flowing into your program: reading from a file on disk with a `FileInputStream`, taking user input from `System.in`, or input from a network socket.

`OutputStream` objects represent data sinks, places we can write data to: `FileOutputStream`s for saving to files, `System.out` is a `PrintStream` or the output to a network socket.

20.4 Blocking

Blocking means that a thread waits (without doing further work) until an event occurs.

Socket input/output streams exhibit blocking behavior:

- When an incoming socket's buffer is empty, calling `read` blocks until data are available.
- When the destination socket's buffer is full, `write` blocks until space is available.

Concurrent modules don't work in lockstep like sequential programs do, so they typically have to wait for each other to catch up when coordinated action is required.

Deadlock can be caused because of it: modules are waiting for each other to do something, so none of them can make any progress.

Notice how both `ServerSocket.accept()` and `in.readLine()` are blocking. This means that the server will need a new thread to handle I/O with each new client. While the client-specific thread is working with that client (perhaps blocked in a read or a write), another thread (perhaps the main thread) is blocked waiting to accept a new connection.

20.5 protocol

A protocol is a set of messages that can be exchanged by two communicating parties. A wire protocol in particular is a set of messages represented as byte sequences, like hello world and bye

21 Shared Memory Programming

Reasons for using multithreading:

- **Parallelism:** a parallel algorithm needs to have different threads accessing the same data structures in an unpredictable way.
- **Responsiveness:** Many programs, including operating systems and programs with user interfaces, want/need to respond to external events quickly. One way to do this is to have threads doing the program's expensive computations while other threads are responsible for "listening to" events like buttons being clicked or typing occurring. The listening threads can then (quickly) write to fields that the computation threads later read.
- **Processor utilization:** It takes a long time for data to be fetched either from disk or outer source. Instead of stalling and waiting, it might be better to allow a different thread to operate. Via having other threads, the program can do useful work while waiting for I/O. This use of multithreading is called **masking** (or **hiding**) I/O latency.
- **Failure/performance isolation:** Other threads can continue executing if one thread throws exceptions or errors due to some problems.

21.1 Synchroniztion

To make sure that threads do not cause data exposure (accessing data at the same time which breaks the operation), we need to enforce **mutual exclusion**

We can do so by:

- have operations to be done "all-at-once" or **atomically** so that we ensure operations are all independent and not interruptable.
- To get out of this conundrum, we rely on **synchronization primitives** provided by the programming language. These rely on special hardware instructions that can do more all at once: these instructions allow us to do more than read or write to a memory address.
- That is, they give us the **atomic** check-for-no-sign-and-hang-a-sign that we want. This is enough to implement mutual exclusion properly. The particular synchronization mechanism we will use is **locking**, and the special hard-

ware instructions involved are "compare-and-set" or "compare-and-swap" instructions.

21.2 Locks

We can define a mutual-exclusion lock, also known as a lock (or sometimes called a mutex), as an abstract datatype designed for concurrent programming and supporting three operations:

- new: establish a new instance of lock, unheld
- acquire: takes a lock and blocks until "not held"
- release: takes a lock and sets it to be "not held"

Held locks would only allow the thread that owns the lock to access the data until the lock is released. However, we should implement a lock in a way that could handle multiple "acquire" operations on one "unheld" lock at the same time. One should win and others should be blocked.

Analyze the following code:

```
class BankAccount {  
    private int balance = 0;  
    private Lock lk = new Lock();  
    ...  
    void withdraw(int amount) {  
        lk.acquire(); /* may block */  
        int b = getBalance();  
        if (amount > b)  
            throw new WithdrawTooLargeException();  
        setBalance(b - amount);  
        lk.release();  
    }  
    // deposit would also acquire/release lk  
}
```

There are a few problems with the code:

1. if $\text{amount} > b$, an exception is thrown without releasing the lock, making other threads not able to access.

2. Notice that the method calls other methods while holding the lock. These methods might also be used from other places which could be locked elsewhere. This way everything will stall! There are two possible solutions:

- There are two solutions to this problem. The first solution is more difficult for programmers: we can define two versions of `getBalance` and `setBalance`. One would acquire the lock and one would assume the caller holds it. The latter could be private, though there are any number of ways to manually keep track in the programmer's head (and through documentation!) of what locks the program holds where.
- The second solution is to change our definition of the acquire and release operations so that it is okay for a thread to (re)acquire a lock it holds. Locks that support this are called reentrant locks. (The name fits with our do-not-disturb analogy.) To define reentrant locks, we need to adjust the definitions for the three operations by adding a count to track how many times the current holder has reacquired the lock (and to maintain the identity of the current holder):
 - new creates a new lock with no current holder and a count of 0
 - acquire blocks if there is a current holder different from the thread calling it. Else if the current holder is the thread calling it, do not block and increment the counter. Else there is no current holder, so set the current holder to the calling thread.
 - release only releases the lock (sets the current holder to “none”) if the count is 0. Otherwise, it decrements the count.

In other words, a lock is released when the number of release operations by the holding thread equals the number of acquire operations.

21.2.1 **synchronized** keyword

```
synchronized ( expression ) {  
    statements  
}
```

- The expression is evaluated. It must produce a reference to an object — not null or a number. This object is treated as a lock. All objects in Java can be locked.

- The synchronized statement acquires the lock
- After the lock is successfully acquired, the statements are executed.
- When control leaves the statements, the lock is released. This happens either when the final ‘}’ is reached or when the program ”jumps out of the statement” via an ‘exception’, a ‘return’, a ‘break’, or a ‘continue’.

Now this way, it handles nearly all scenarios for you.

Although we can pass any object to synchronized, it might be better to just directly call on the object that holds the whole field like:

```

        void withdraw(int amount) {
            synchronized (this) {
                int b = getBalance();
                if (amount > b) {
                    throw new WithdrawTooLargeException();
                }
                setBalance(b - amount);
            }
        }

        void doubleBalance(BankAccount acct) {
            synchronized(acct) {
                acct.setBalance(acct.getBalance()*2);
            }
        }

```

Java also holds another feature that allows you to directly synchronize the method:

```

        class BankAccount {
            private int balance = 0;

            synchronized int getBalance() {
                return balance;
            }

            synchronized void setBalance(int x) {
                balance = x;
            }
        }

```

```

    }

    synchronized void withdraw(int amount) {
        int b = getBalance();
        if(amount > b) {
            throw new WithdrawTooLargeException();
        }
        setBalance(b - amount);
    }
    // deposit and other operations would also be declared synchronized
}

```

It is the same as synchronizing ‘this’ object. But might be easier to read and use.

Warning: locks work on objects instead of the pointer. So if we change the pointer of a synchronized object to a new object, then the new object is not synchronized!!!!!!!!!!!!!!

22 Data Races

A *race condition* is a mistake in your program such that whether the program behaves correctly or not depends on the order in which the threads execute.

22.1 Bad Interleaving

One kind of race condition is a bad interleaving, sometimes called a higher-level race. If doing something at the same time as doing another thing causes dependency, then it is bad interleaving.

A data race is a specific kind of race condition that is better described as a *simultaneous access error*. There are two kinds of data races:

- When one thread might read an object field at the same moment that another thread writes the same field.
- When one thread might write an object field at the same moment that another thread also writes the same field.

Reading at the same time is ok.

22.2 volatile

Instead of using locks, we can declare fields to be volatile. By definition, accesses to volatile fields do not count as data races, so programmers using volatiles are still upholding their end of the grand compromise.

Reading/writing a volatile field is less efficient than reading/writing a regular field, but more efficient than acquiring and releasing a lock.

Volatile fields are typically used only by concurrency experts. They can be convenient when concurrent code is sharing only a single field. Usually with more than one field, you need critical sections longer than a single memory access, in which case the right tool is a lock, not a volatile field.

23 Thread Safety

Recall race conditions: multiple threads sharing the same mutable variable without coordinating what they're doing. This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

There are basically four ways to make variable access safe in shared-memory concurrency:

- **Confinement.** Don't share the variable between threads. This idea is called confinement.
- **Immutability.** Make the shared data immutable. Some additional constraints for concurrent programming are different from previous sections.
- **Thread-safe data types.** Encapsulate the shared data in an existing thread-safe data type that does the coordination for you.
- **Synchronization.** Use synchronization to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own thread-safe data type.

A data type or static method is thread-safe if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.

23.1 Confinement

You avoid races on mutable data by keeping that data confined to a single thread. Do not give any other threads the ability to read or write the data directly. Since shared mutable data is the root cause of a race condition, confinement solves it by not sharing the mutable data.

Pay attention to static/global variables and references! Maybe they will be exposed to other threads without even noticing.

23.2 Immutability

An immutable type is generally safe.

An immutable data type that uses beneficent mutation will have to make itself thread-safe using locks

So in order to be confident that an immutable data type is thread-safe without locks, we need a stronger definition of immutability:

- no mutator methods;
- all fields are private and final;
- no mutation whatsoever of mutable objects in the rep, not even beneficent mutation;
- no rep exposure (see the ADT reading to review what rep exposure is).

23.3 Thread-safe Data Types

When a data type in the Java library is Thread-safe, its documentation will explicitly state that fact.

Note that `StringBuffer` is a thread-safe data type whereas `StringBuilder` is not. Can be found in their specs.

`Collections.synchronizedMap(new HashMap<>())` is another example of a synchronized wrapper. Now the operations are safe but iterators or for loops are still not safe.

The solution to this problem will be to acquire the collection's lock when you need to iterate over it, which we'll talk about in a subsequent reading.

However, it is still not enough:

```
if (!lst.isEmpty()) {String s = lst.get(0); ... }
```

It is not safe since another thread may have removed an element between the if statement and the line of execution, even if the data type of 'lst' might be safe.

23.4 Thread Safety Argument

A safety argument needs to catalog all the threads that exist in your module or program, and the data that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: confinement, immutability, Thread-safe data types, or synchronization. When you use the last two, you also need to argue that all accesses to the data are appropriately atomic: that is, that the invariants you depend on are not threatened by interleaving.

24 Guidance for Safe Concurrent Programming

Every memory location in your program should meet at least one of the following:

- Thread-local: Only one thread ever accesses it.
- Immutable: After initialization, only read not written.
- Synchronized: Locks are used to ensure there are no race conditions.

Synchronization is hard, so try to use the other two as much as possible.

24.1 General Guidance

Lack of synchronization is usually caused by two:

- Data races: simultaneous read/write or write/write of the same memory location.
- Bad interleavings: exposing bad intermediate state, which can be easily interrupted by another thread

24.1.1 Three choices to fix it

For every memory location in the program, at least one of the following must be obeyed:

1. Thread-local: Do not use the location in more than 1 thread.

2. Immutable: Do not write to the memory location.
3. Synchronized: Use synchronization to control access to the location.

24.1.2 Guidance on locking mechanism

1. Eliminate data races: Never allow two threads to read/write or write/write the same location at the same time.
2. Consistent Locking: For each location needing synchronization, have a lock that is always held when reading or writing the location. This could avoid data racing but can not solve bad interleaving.

24.1.3 Lock Granularity

Two styles of using locks:

1. Coarse-grained: fewer locks, more objects per lock.
 - simpler to implement
 - faster access to multiple locations
2. Fine-grained: more locks, fewer objects per lock.
 - More simultaneous access. May be facing less unnecessary blocking.

General advice is to start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue. Alas, often leads to bugs.

24.1.4 Atomicity

An operation is atomic if no other thread can see it partly executed.

Make critical sections just long enough to preserve atomicity. Then design the locking protocol to implement the critical sections correctly.

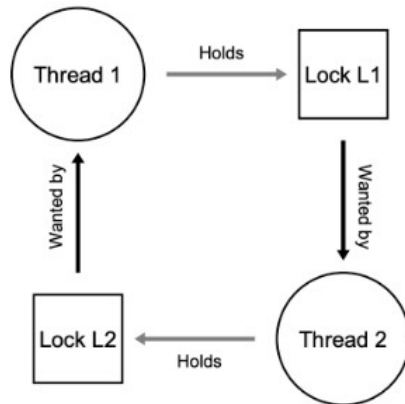
Atomic Variables are available in Java: “AtomicInteger, AtomicLong, AtomicBoolean, and AtomicReference” with methods:

- get() gets the value from the memory, so that changes made by other threads are visible; equivalent to reading a volatile variable

- `set()` writes the value to memory so that the change is visible to other threads; equivalent to writing a volatile variable
- `lazySet()` eventually writes the value to memory, which may be reordered with subsequent relevant memory operations. One use case is nullifying references, for the sake of garbage collection, which is never going to be accessed again. In this case, better performance is achieved by delaying the null volatile write
- `compareAndSet()` returns true when it succeeds, else false
- `weakCompareAndSet()` weaker in the sense, that it does not create happens-before orderings. This means that it may not necessarily see updates made to other variables

25 Deadlock

A deadlock is a situation in which two or more threads are blocked forever, each waiting on the other.



25.1 Conditions for Deadlock

Deadlock cannot occur if any of the below conditions is not satisfied

- Circular wait
- Mutual exclusion — > Thread needs to acquire the lock exclusively

- Hold-and-wait — > Thread holds a lock and waits for another lock
- No preemption — > Thread cannot be forced to release a lock

25.2 Preventing Deadlock

Look at an example buggy code and try to fix it:

```
public class Account {
    private int balance = 0;
    public void deposit(int amount) { balance += amount;}
    public void withdraw(int amount) { balance -= amount;}
    public int getBalance() { return balance; }
    public void transfer(Account target, int amount) {
        synchronized (this) {
            withdraw(amount);
            synchronized (target) {
                target.deposit(amount);
            }
        }
    }
}
```

25.2.1 Preventing Circular Wait

Providing a **total ordering** of all resources and requiring that each thread acquire resources in increasing order of enumeration.

An possible fix example:

```
public void transfer(Account target, int amount) {
    Account first = this, second = target;
    if (System.identityHashCode(this)
        > System.identityHashCode(target)) {
        first = target;
        second = this;
    }
    synchronized (first) {
        synchronized (second) {
            withdraw(amount);
        }
    }
}
```

```

        target.deposit(amount);
    }
}

```

Note that it works since there is always an ordering of the two accounts. No transaction would hold the second lock before the first one. (hashCode is used just for demonstration purposes, hashCode is not guaranteed to be unique)

25.2.2 Preventing Hold-and-Wait

Can be avoided by acquiring all the locks at once.

However, it requires knowing which locks must be held and to acquire them ahead of time. Then all locks must be acquired early on (at once) instead of when they really need to be used. This could decrease concurrency and performance.

This acquiring at the same time mechanism can be achieved by having a global lock that is acquired before any other locks.

Possible fix:

```

public void transfer(Account target, int amount) {
    synchronized (globalLock) {
        synchronized (this) {
            synchronized (target) {
                withdraw(amount);
                target.deposit(amount);
            }
        }
    }
}

```

I would say do not do this unless it is the only option. It is not a good idea to have a global lock. It is not scalable and it is not flexible. We use multithreading to increase performance and concurrency. This way, we are just making it sequential.

25.2.3 Preventing No Preemption

There is a tryLock() method that returns true if the lock is acquired and false if it is not. But it can also take arguments to set a timeout.

```

boolean tryLock()
boolean tryLock(long time, TimeUnit unit)

```

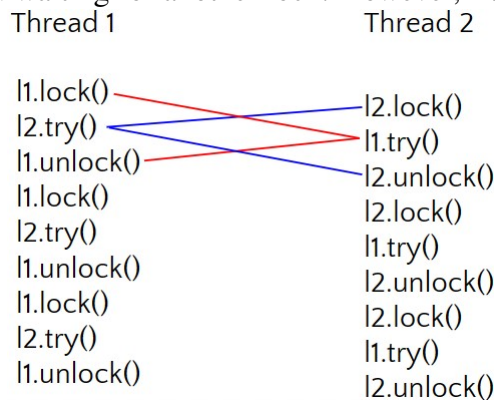
Possible fix:

```

boolean retry = true;
while (retry) {
    lock1.lock()
    if (lock2.trylock() == false){
        lock1.unlock();
    } else { // lock2 is acquired
        try {
            // exec the critical section
        } finally {
            lock2.unlock();
        }
        lock1.unlock();
        retry = false;
    }
}

```

Here we use a while loop to keep trying to acquire the lock. If it fails, we release the lock and try again. This way, we can avoid deadlock by not holding the lock and waiting for another lock. However, **livelock** will be introduced.



Here, it is possible that l1 and l2 keep trying to acquire the lock and keep failing. This is livelock. It locks itself and tries to lock the other one. If the frequency of the two are exactly matching, then it will be livelock.

Livelock:

- Occurs when two or more interacting threads are stuck in a loop of repeated actions
- Unlike a deadlock, entities remain active but cannot progress

- The perpetual cycle of actions and reactions can be considered analogous to two people attempting to pass each other in a corridor: Alphonse moves left, so Bertrand moves left to compensate. Meanwhile, Alphonse moves right to compensate Bertrand's movement, and so on.
- One possible solution would be to add a randomized delay to break patterns like setting up a random wait time for 'tryLock()'

25.2.4 Mutual Exclusion

The best to prevent deadlocks is to avoid locks if possible. Think carefully about whether you really need to lock the object.