CS2030S Programming Methodology II

Ryan Joo

AY25/26 Sem 1

Abstract

Lecture notes: https://nus-cs2030s.github.io/2526-s1/

Contents

revision tips: read notes properly, understand it intuitively go thru recitations, identify learning points
PYP: do under exam conditions (timed)

1 Variables and Types

Variable: abstraction that allows us to give a user-friendly name to data in memory
Type of a variable decides the operations that can be performed on a variable

- **Statically typed** (Java)
  - Variable can only hold values of the same type (assigned at compile time)
    Compile time type: type the variable is assigned with when declaring the variable - used by the compiler when parsing syntax and checking for type mismatches
  - Type checking during compile time: compiler checks if compile-time type matches when it parses code - throws an error if there is type mismatch

  Dynamically typed
  - Same variable can hold values of different types
  - Type checking during run time: type mismatch error is only caught during run time
- **Type system**: a set of rules that governs how the types can interact with each other
  Strongly typed: Enforces strict rules in type system to ensure type safety. Catches type errors during compile time rather than leaving it to runtime.
  Weakly typed: More permissive in terms of typing checking, e.g. can convert string to integer.
- **Primitive types**: primitive variable stores values (numeric, boolean)
  - Primitive variables never share their value with each other
  - E.g. `byte`, `short`, `int`, `char`, `long`, `float`, `double`; `boolean`
- **Reference types**: reference variable stores the reference to the value
  - Two reference variables can share the same value, since they refer to the same object in memory → changing the field of one causes the field of the other to change as well
  - E.g. `String` - reference variables are practically always objects in Java
- T is a **subtype** of S, denoted by $T <: S$, if a piece of code written for variables of type S can also be safely used on variables of type T. (S is a **supertype** of T.)
  (i) **Reflexive**: For any type S, $S <: S$
  (ii) **Transitive**: If $S <: T$ and $T <: U$, then $S <: U$
  (iii) **Anti-symmetric**: If $S <: T$ and $T <: S$, then $S = T$

$$\begin{tikzcd}
\texttt{byte} \arrow[r] & \texttt{short} \arrow[r] & \texttt{int} \arrow[r] & \texttt{long} \arrow[r] & \texttt{float} \arrow[r] & \texttt{double} \\
 & \texttt{char} \arrow[ur] & & & &
\end{tikzcd}$$

- **Type casting**: assign a value of one data type to another type
  - **Widening type conversion**: if $S <: T$, then a variable of type T can automatically hold a value from a variable of type S

```
T x = S y;   // ok
```

  - **Narrowing type conversion**: convert variable of type S to T if $T <: S$
    Requires explicit typecasting (if not, code won't compile)

    ```
    T x = (T) y; // explicit typecast of y to T
    ```

    If runtime type of target is not the same as the cast type, then runtime error will occur

2 Functions

- **Function**: abstraction that groups a set of instructions and gives them a name
- **Method**: Java terminology for function
- **Abstraction barrier**: separates the role of the programmer into two: implementer & client
  - Above the barrier: **implementer** provides implementation of abstraction
  - Below the barrier: **client** uses abstraction to perform task

  This enforces separation of concerns between roles

Part I

Object-Oriented Programming

3 Encapsulation

- **Composite data type**: group primitive types together using a name
- **Class**: bundle composite data type & associated functions on the same side of abstraction barrier
  1. **Fields**: data in the class
  2. **Methods**: functions in the class
- **Encapsulation**: keep all the data and functions operating on the data related to a composite data type together within an abstraction barrier
- **Object**: instance of a class
- Classes are reference types $\rightarrow$ **aliasing** (see stack and heap)

```
Circle c1 = new Circle();
Circle c2 = c1; // any changes to c2 affects c1 and vice versa
```

Information Hiding

Access modifiers:

Accessed from	`private`	`public`
Inside the class (same class)	✓	✓
Outside the class (another class)	×	✓

Information hiding: protect abstraction barrier from being broken by explicitly specifying if a field/method can be accessed from outside the abstraction barrier

- Private fields: prevent someone from changing the values arbitrarily - implementation details should be hidden from the client
- Public methods: any functionalities should be codified as API
- Enforced by compiler at compile time

Does the program follow the principle of information hiding?
No. The field `balance` in `BankAccount` is publicly accessible.

"Tell, Don't Ask" principle:

- Don't ask an object for its state (using getters/setters), then perform the task on its behalf
- Tell an object what to do – a task that is performed only on the fields of a class should be implemented in the class itself (**responsibility**)

Does this program follow the principle of "Tell, Don't Ask"?

No. The `Customer` class directly asks for the field `balance` in `BankAccount` to check directly in `Customer` class instead of telling the `BankAccount` class to check if there is enough balance.

The `Customer` class also asks for the value of the field `balance` in `BankAccount`, modifies it, and stores it back, instead of telling `BankAccount` class to update its own `balance`.

Class Fields

- **Class field**: associated with a class (exactly one instance throughout lifetime of the program)
  - Declare using `static`
  - Accessed through class name (without instantiating class)
- **Instance field**: associated with an object

Class Methods

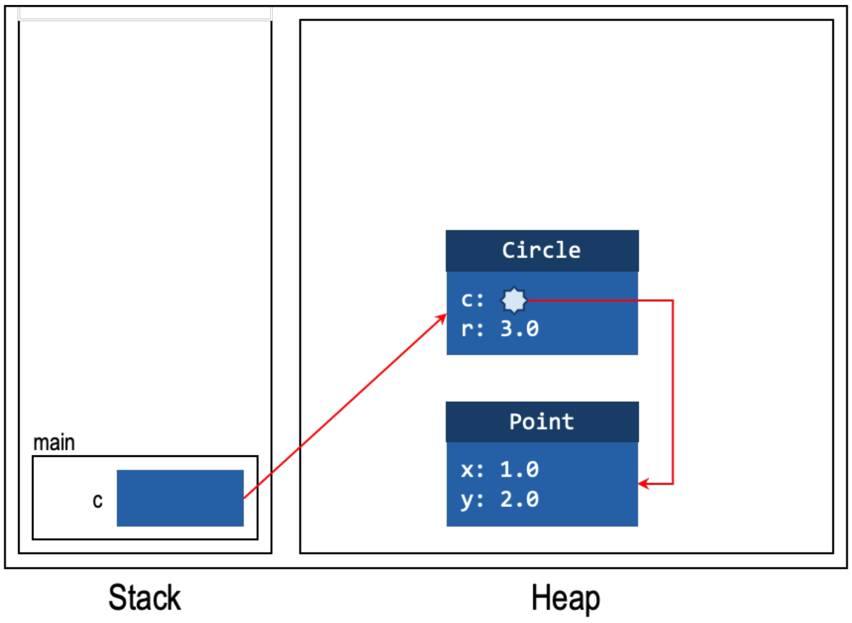
- **Class method**: associated with a class
  - Invoked without an instance, so no access to the instance's (non-static) fields or methods $\rightarrow$ `this` has no meaning
  - Accessed through class name (without instantiating class)
- **Instance method**: defined inside of a class - varies with different instances of the class

4 Composition

Composition: HAS-A relationship (use other class as a field)

```
class Circle {
  Point c;
  double r;
}
```

5 Stack and Heap



- **Stack**: one method invoked ⇒ one **frame** created (`main` frame always exists)
  - Contains **variables** (primitive, reference) & values
  - Instance method: 1. `this` reference 2. method args 3. local variables within the method
    Class method: does not contain `this` reference
  - Denote uninitialised variables by ∅
  - Last-In First-Out (LIFO): elements can can only be added or removed from the top
  - When the method completes, the frame is removed
- **Heap**: one `new` keyword ⇒ one **object** created
- **Metaspace**: static fields of classes
- **Aliasing**: two different arrows point to the same location (reference types share the same reference value)

6 Inheritance

Inheritance: IS-A relationship (`extends` creates subtype relationship)

- Subclass **inherits** all accessible fields/methods from superclass
  - Can access all public fields/methods of superclass
  - Cannot access any private fields/methods of superclass
- Type assignment: **RTT** <: **CTT**

```
// ColoredCircle <: Circle
ColoredCircle c = new Circle(p, 0);       // error, narrowing
Circle c = new ColoredCircle(p, 0, blue); // OK, widening
```

7 Overloading

Method overloading: multiple methods in the same class have same name but different method signature (change type/order/number of parameters)

```java
public int add(int x, int y) {
  return x + y;
}
public int add(int x, int y, int z) {
  return x + y + z;
}
```

If two methods have same name & method signature, they are not overloaded → cannot compile

8 Polymorphism

Method Overriding

- **Method signature** = method name + number, type, order of parameters

  Method descriptor = method signature + return type

  Method signature: `C::foo(B1, B2)`
  Method descriptor: `A C::foo(B1, B2)`

- **Method overriding**: subclass' instance method has same method descriptor as superclass

  Instance method in subclass **overrides** instance method in superclass

  ```java
  class Circle {
    @Override
    public String toString() {
      return "{ center: " + this.c + ", radius: " + this.r + " }";
    }
  }
  ```

Polymorphism

Polymorphism: subclasses define their own unique behaviours, yet share some of the same functionality of superclass
Using **method overriding**, the same target of invocation can invoke different methods

1. Identify types involed. Create a (potentially abstract) class for each type (if not already).
2. Identify common superclass or create one if not already present. Create a (potentially abstract) common method in the superclass.
3. Override the common method in subclass.

Dynamic Binding

Recall CTT and RTT:

```java
A a = new B(); // CTT(a) = A, RTT(a) = B
```

Dynamic binding: decide which *instance* method is invoked. For example, consider `curr.foo(arg)`

Compile time step:

1. Determine CTT(`curr`)
2. In CTT(`curr`), find all accessible methods with name `foo`
3. Check `arg` can bind to which method, i.e., CTT(`arg`) <: CTT(`param`)
4. Choose the most specific one (use subtyping relationship)

   Record its method descriptor M

Run time step:

1. Determine RTT(`curr`)
2. In RTT(`curr`), find M *exactly*
3. Found it? Execute M

   No find? Repeat search in the superclass

LSP

Liskov Substitution Principle: Let $\phi(x)$ be a property provable about objects x of type T. If $S <: T$, then $\phi(y)$ should be true for objects y of type S.

- Informally, if $S <: T$, then an object of type T can be replaced by that of type S without changing the **desirable property** of the program
- A subclass should not break the expectation of the superclass

Is LSP violated?

Yes, it violates LSP. The subclass changed the behaviour of the superclass, so the property that [insert desirable property] no longer holds for the subclass. Places in a program where the superclass is used cannot be simply replaced by the subclass.

`final` : used to prevent

- field from being re-assigned (one assignment when constructor invoked)
- method from being overidden (**prevent overriding**)
- classes from being inherited (**prevent inheritance**)

Abstract Class

- **Abstract method**: cannot be implemented, and should not have a method body
- **Abstract class**: may provide no/incomplete implementation for its methods

  Cannot be instantiated
- A class with at least one abstract method must be declared abstract.

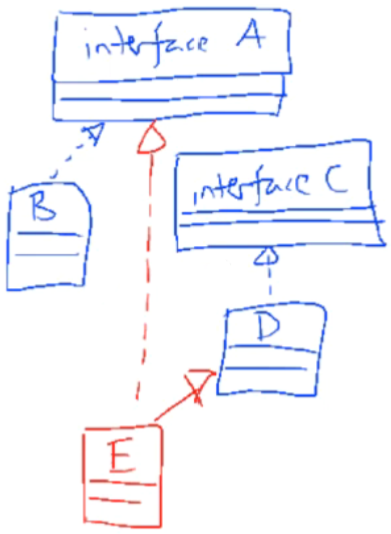
  On the other hand, an abstract class may have no abstract method.
- **Concrete class**: not abstract (no abstract methods)

  Concrete subclass of abstract class must override abstract method(s)

Interface

- **Interface**: models what an entity can do
  - All methods `public abstract`, no fields
  - Implement an interface using `implements`
  - Like abstract classes, interfaces cannot be instantiated
- **Inheritance rules**: if a class C implements an interface I, then $C <: I$
  - A class can extend at most one class (including abstract class)
    A class can implement multiple interfaces
  - An interface cannot extend from another class
    An interface can extend multiple interfaces
- Casting using an interface:

```
interface A { }
class B implements A { }
interface C { }
class D implements C { }
```



```
B b = new B();
D d = (D) b;
```

does not compile since there is no subtype relationship between `B` and `D` (class diagram is not connected).

```
A a = new B();  // ok, B <: A
D d = (D) a;
```

compiles since it is *possible* that a subclass `E` *could* extends `D` and implements `A`.

9 Types

Wrapper Class

Wrapper class: a class that encapsulates a primitive type (allows us to treat primitive types as reference types)

E.g. `Integer` for `int`, `Double` for `double`

- Wrapper objects are immutable (cannot be changed after creation)
- **Auto-boxing**: primitive to wrapper
  Unboxing: wrapper to primitive

```
Integer i = 4; // auto-boxing
int j = i;     // unboxing
```

- No subtyping between wrappers

  ```
  Double d = 4; // Invalid: 4 is autoboxed to Integer, but Integer /<: Double
  Object o = 4; // Valid: 4 is autoboxed to Integer, and Integer <: Object
  ```

- **Added performance costs**: Due to immutability, during every mutating operation, a new wrapper object is created

  ```
  Double sum = 0.0;
  for (int i = 0; i < Integer.MAX_VALUE; i++) {
    sum += i; // every time sum is updated, a new Double object is created
  }
  ```

Runtime Class Mismatch

Narrowing type conversion requires explicit casting.
Typecast check: how do we know when we can cast? For example, consider `a = (C)b`

Compile-time check:

1. Check if it is *possible* for RTT(`b`) $<:$ `C`. If impossible, throw compilation error.
2. Check if `C` $<:$ CTT(`a`). If impossible, throw compilation error.

Runtime check:

3. Check if RTT(`b`) $<:$ `C`. If not, throw a runtime error (runtime class mismatch).

Variance

Let $C(T)$ be a complex type (e.g. arrays) based on type T. We say that C is

- covariant if $S <: T \Rightarrow C(S) <: C(T)$ (preserve subtyping relationship)
- contravariant if $S <: T \Rightarrow C(T) <: C(S)$ (flip subtyping relationship)
- **invariant** if it is neither covariant nor contravariant

Java arrays of reference types are **covariant**: $S <: T \Rightarrow S[\,] <: T[\,]$
Possibility of runtime error:

```
Integer[] intArray = new Integer[2] {Integer.valueOf(10), Integer.valueOf(20)};
Object[] objArray;
objArray = intArray;  // ok, since Integer <: Object
objArray[0] = "Hello!"; // ok, since String <: Object
// compiles!
```

Compiler does not know that `objArray` is referring to an `Integer[]` until runtime – only then would Java realise that we are trying to stuff a string into an array of integers!

10 Exceptions

Handling Exceptions

`try`-`catch`-`finally` block

```
try {
  // do something
} catch (an exception parameter) {
  // handle exception
} catch (another exception parameter) {
  // can have more catch blocks
} finally {
  // clean up code
  // regardless of there is an exception or not
}
```

When there are multiple catch blocks,

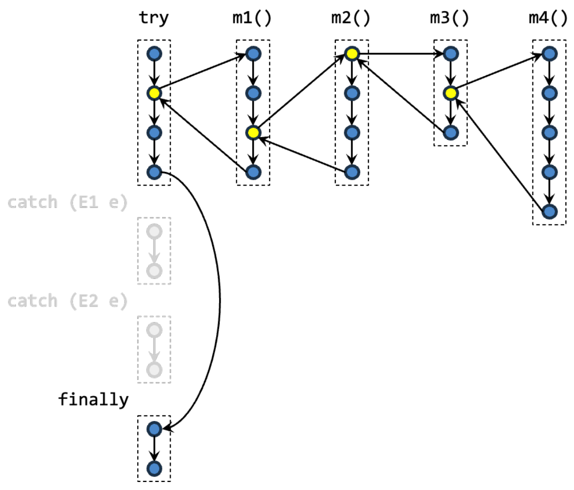
1. Check in the order they appear
2. Check subtyping relationship: choose the first `catch` block that the thrown exception can bind to

If `ExceptionX` <: `ExceptionY`, the second catch will never be executed → prevented with compilation error

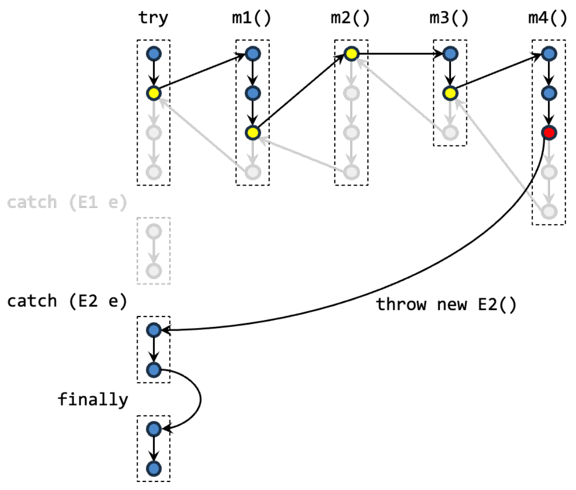
```
  :
} catch(ExceptionY e) {
  // handle ExceptionY
} catch(ExceptionX e) {
  // handle ExceptionX
}
  :
```

Control Flow of Exceptions

No exception:



Exception:



Checked vs Unchecked Exceptions

Checked exception <: `Exception`: programmer has no control over, even if perfect code is written → actively anticipate the exception and handle them

- Exceptions that are checked at compile time

- Compiler forces you to either handle them with a `try`-`catch` block, or declare them in the method signature with `throws` (else program won't compile)
- E.g. `FileNotFoundException`
- Represent conditions you can reasonably expect and recover from

```java
class InvalidCircleException extends Exception {
  public InvalidCircleException(double r) {
    super("The radius: " + r + " should not be negative.");
  }
}

class Circle {
  public Circle(Point c, double r) throws
      InvalidCircleException {
    if (r < 0) {
      throw new InvalidCircleException(r);
    }
  }
}
```

```java
try {
  Circle c = new Circle(new Point
      (0.0), -1);
} catch (InvalidCircleException e) {
  System.out.println(e);
}
```

Unchecked exception <: `RuntimeException` : caused by programmer's errors, should not happen if perfect code is written

- Exceptions that are not checked at compile time
- Compiler does not force you to catch or declare them → propagate down the stack → error message
- E.g. `IllegalArgumentException` , `NullPointerException` , `ClassCastException`
- Indicate programming errors which cause runtime errors

```java
class InvalidCircleException extends RuntimeException {
  public InvalidCircleException(double r) {
    super("The radius: " + r + "should not be negative.");
  }
}

class Circle {
  public Circle(Point c, double r) {
    if (r < 0) {
      throw new InvalidCircleException(r);
    }
  }
}
```

Override a method that throws a checked exception:

- Overriding method must throw same/more specific exception than the overridden method
- In line with LSP: caller of the overridden method cannot expect any new checked exception beyond what has already been "promised" in the method specification

11 Generics

- **Generic class**: takes other types as type parameters

  ```
  class Box<T1, T2, ..., Tn> { /* ... */ }
  // type parameters T1, T2, ..., Tn (only reference types)
  ```

- An invocation of a generic type is called a **parameterised type**

  ```
  Box<Integer> integerBox;
  ```

- **Diamond operator**: type arguments inferred as CTT (see Type Inference)

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

- **Extend generics**: `String` is fixed, `T` can be any type we want for `DictionaryEntry`

  ```
  class DictEntry<T> extends Pair<String,T> {
    :
  }
  ```

- **Generic method**: parametrise a method with type parameters, without being in a generic class

  ```
  class A {
    public <T> boolean contains(T[] array, T obj) {
      // :
    }
  }
  ```

  Calling a generic method: `A.<String>contains(strArray, "123")`

- **Bounded type parameter**: constrain type parameter to only be subtypes, using `extends`

  ```
  <T extends GetAreable> T findLargest(T[] array) { ... }
  ```

  If `T` is *at most* a `GetAreable`, then `T` must have `getArea()`

- Generics are **invariant**: no subtyping relationship

Type Erasure

Type erasure procedure:

1. Remove angle brackets, replace type parameters with their bounds (`Object` if unbounded)
2. If generic type is instantiated and used, insert type casts to preserve type safety

   ```
   // Integer i = new Pair<String,Integer>("hello", 4).getSecond();
   Integer i = (Integer) new Pair("hello", 4).getSecond();
   ```

3. Generate bridge methods to preserve polymorphism in extended generic types

   (After type erasure, method signatures of method in subclass & superclass don't match → subclass method does not override superclass)

   ```
   public int compareTo(Object var1) {
     return this.compareTo((Pair) var1); // delegate to compareTo(Pair) in subclass
   }
   ```

Generics and Arrays Can't Mix

Generic array **declaration** is OK, **instantiation** is not OK.

```java
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2]; // instantiation not OK
new Pair<S,T>[2];                                               // instantiation not OK
new T[2];                                                       // instantiation not OK
T[] array;                                                      // declaration is OK
```

Reason:

```java
// create a new array of pairs
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];
// pass around the array of pairs as an array of object
Object[] objArray = pairArray;
// put a pair into the array - no ArrayStoreException, since Java arrays are covariant
objArray[0] = new Pair<Double,Boolean>(3.14, true);
```

After type erasure,

```java
Pair[] pairArray = new Pair[2];
Object[] objArray = pairArray;
objArray[0] = new Pair(3.14, true);
```

Heap pollution: a variable of parameterised type refers to an object not of that parameterised type $\rightarrow$ retrieving that variable leads to `ClassCastException`

Java arrays are **reifiable**: full type information is available during runtime (Java runtime checks if items in array matche the type of array $\rightarrow$ throw `ArrayStoreException` if mismatch)

Java generics are **not reifiable**: due to type erasure, type information missing during runtime

Unchecked Warnings

```java
// Create an array with type parameters
class Seq<T> {
  private T[] array;

  public Seq(int size) {
    // The only way we can put an object into array is through set(), and we only put object of
        type T inside. Safe to cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }

  /**
  public T[] getArray() {
    return this.array;
  }
  */
}
```

Note that `getArray()` is unsafe due to the following:

```
Seq<String> strSeq = new Seq<String>(2);
Object[] objArray = strSeq.getArray();
objArray[0] = 5;
String s = strSeq.get(0);
```

Unchecked warning: message from the compiler that there could be runtime error that it cannot prevent, due to type erasures

- `Seq` is generic with type parameter `T` and no upper bound.
- After type erasure, `T` is replaced with `Object`.
- Compiler can't verify at runtime that `Object[]` will only contain `T` objects, since erasure removes the type info needed for this check.

  ```
  Seq<String> seq = new Seq<String>(4);
  Object[] objArray = seq.getArray();
  objArray[0] = 4;
  seq.get(0); // ClassCastException
  ```

- Hence the cast `(T[]) new Object[size]` is unchecked.

`@SuppressWarning("unchecked")` annotation: suppress warning messages from compiler & assure compiler that the type operation is deemed to be safe

- Since `array` is private, the only way to put something in is using `set`
- Since `set` only accepts `T`, objects in `array` must be of type `T`
- Casting `Object[]` to `T[]` is type-safe

Raw Types

Raw types: generic type used without type arguments → use whatever the type erased version is and compile from there

```
Box rawBox = new Box(); // compiles, even if type T is not specified
```

Since `rawBox` is raw, compiler doesn't know what `T` is → treats `T` as `Object` (erased upper bound)

- For backward compatibility, assigning a parameterised type to its raw type is allowed:

  ```
  Box<String> stringBox = new Box<>();
  Box rawBox = stringBox;           // OK
  ```

  This is safe because `rawBox` can hold anything (`Object`).

- But if you assign a raw type to a parameterised type, you get a warning:

  ```
  Box rawBox = new Box();        // rawBox is a raw type of Box<T>
  Box<Integer> intBox = rawBox;  // warning: unchecked conversion
  ```

  Due to type erasure, compiler cannot verify that `rawBox` (which could contain `Object`) actually holds only `Integer` objects → possible `ClassCastException` if you later retrieve a non-`Integer`.

- You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

  ```
  Box<String> stringBox = new Box<>();
  Box rawBox = stringBox;
  rawBox.set(8); // warning: unchecked invocation to set(T)
  ```

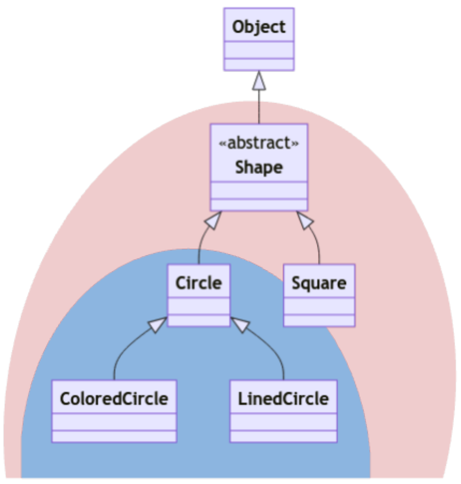
  `set(T)` expects a specific type (e.g. `String`), but since `rawBox` is raw, the compiler can't check if `8` (an `int`) matches `T` $\rightarrow$ possible runtime error when you try to retrieve the contents

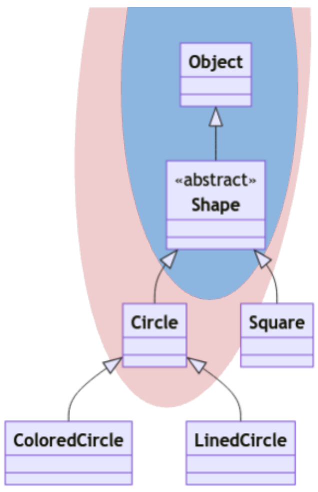
The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

Wildcards

Wildcard: can be substituted for any type (not bound to the declared type parameters)

- **Unbounded wildcard**: `C<?>` means that there exist a type (we do not care about the name, but there is), such that we can use it to replace `?`
  - Cannot use `?` as a type: `? x = ...`, `<?> x = ...` are not allowed
  - For any type `T`, `C<T>` $<:$ `C<?>` (since we can instantiate `C<?> x = new C<T>()`)
- **Upper-bounded wildcard**: `C<? extends T>` means there exist a type that is a subtype of `T`, such that we can use it to replace `?`
  - If `S` $<:$ `T`, then `C<? extends S>` $<:$ `C<? extends T>` (**covariant**)
  - For any type `T`, `C<T>` $<:$ `C<? extends T>`
- **Lower-bounded wildcard**: `C<? super T>` means there exist a type that is a supertype of `T`, such that we can use it to replace `?`
  - If `S` $<:$ `T`, then `C<? super T>` $<:$ `C<? super S>` (**contravariant**)
  - For any type `T`, `C<T>` $<:$ `C<? super T>`





- **Producer Extends; Consumer Super** (PECS) rule:
  - **Producer**: returns a variable of type `T` (e.g. get method), so its type parameter must be subtype of `T` $\rightarrow$ declare with `? extends T`

    - **Consumer**: takes in variable of type `T` (e.g. set method), so its type parameter must be supertype of `T` $\rightarrow$ declare with `? super T`

- **Raw types**: Not allowed; use unbounded wildcards as an alternative

  Instanceof:

  ```
  a instanceof A<String> // doesn't work, since type argument String is not available during
      run-time due to erasure
  a instanceof A // not allowed anymore
  a instanceof A<?>
  ```

  Instantiate generic arrays:

  ```
  new Comparable<String>[10] // cannot instantiate generic array directly
  new Comparable[10] // not allowed anymore
  new Comparable<?>[10]
  ```

  Subtyping between unbounded wildcard and raw type:

  ```
  List<?> lst = new List(); // this assignment works
  ```

  so `List<?>` <: `List`

Consider the classes `S1`, `S2`, and `C<T>` such that `S1` <: `S2`. Then we can form a chain:

`C<S1>` <: `C<? extends S1>` <: `C<? extends S2>` <: `C<S2>`

Solve adversarially: imagine the type parameter being instantiated in the *most inconvenient possible way*, and then deduce what the compiler must still accept

Type Inference

Local type inference algorithm:

1. Write down all local type constraints

    - **Target typing**: if output is assigned to something (T is bound to some type)
    - **Argument typing**: is there wildcards used for `T` in the argument
    - **Type parameter bound**: look at declaration of type parameter - does `T` extend/super something?

2. Solve type constraints

3. Choose the most specific one (mentioned or superclass)

    - `Type1` <: `T` <: `Type2`, then `T` is inferred as `Type1`
    - `Type1` <: `T`, then `T` is inferred as `Type1`
    - `T` <: `Type2`, then `T` is inferred as `Type2`

   Ignore the subclasses not specified in the constraints

   Solution may be a superclass of the types in the constraints

Worked example:

```
<T> boolean contains(Seq<? extends T> seq, T obj) { .. }
```

```
// circleSeq :: Seq<Circle> & shape :: Shape
A.contains(circleSeq, shape);
```

1. **Target typing**: nothing
2. **Argument typing**:
   - `Shape` passed into `T` $\Rightarrow$ `Shape` $<:$ `T`
   - `Seq<Circle>` passed into `Seq<? extends T>` $\Rightarrow$ `Seq<Circle>` $<:$ `Seq<? extends T>` $\Rightarrow$ `Circle` $<:$ `T`
3. **Type parameter bound**: nothing

The constraints are `Shape` $<:$ `T` and `Circle` $<:$ `T`, so `T` is inferred as `Shape`.

Part II
Functional Programming

12 Immutability

Immutable class: instance cannot have any visible changes outside its abstraction barrier
Making a class immutable:

1. Make fields `final` to avoid re-assignment

   Remove any assignments to the fields (compilation error due to `final`)

2. Make class `final` to disallow inheritance → avoid subclasses from overriding the methods

3. Change setter from `void` to return a new instance → prevent mutating the current instance

Advantages of Being Immutable

1. **Ease of understanding**:

   As long as a variable is not re-assigned later on, any references to it will always be to the originally instantiated version and no changes will affect this original reference

2. **Enabling safe sharing of objects**:

   Safely share instances of immutable class → reduce the need to create multiple copies of the same object

   The origin $(0, 0)$ is commonly used, so we create and cache a single copy of the origin, always return this copy when the origin is required

   ```java
   final class Point {
     private final static Point ORIGIN = new Point(0, 0); // cache a single copy of origin
     public static Point of(double x, double y) { // factory method
       if (x == 0 && y == 0) {
         return ORIGIN; // always return this copy when the origin is required
       }
       return new Point(x, y);
     }
   }
   ```

3. **Enabling safe sharing of internals**:

   Immutable instances can share their internals freely (since underlying fields will not mutate)

   ```java
   public final class ImmutableSeq<T> {
     private final int start;
     private final int end;
     private final T[] array;

     // Only items of type T goes into the array.
     @SafeVarargs
     public static <T> ImmutableSeq<T> of(T... items) {
       // We need to (explicitly) copy to ensure that it is truly immutable
       @SuppressWarnings("unchecked");
       T[] arr = (T[]) new Object[items.length];
       for (int i = 0; i < items.length; i++) {
   ```

```
    arr[i] = items[i];
  }
  return new ImmutableSeq<>(arr, 0, items.length - 1);
}

private ImmutableSeq(T[] a, int start, int end) {
  this.start = start;
  this.end = end;
  this.array = a;
}

public T get(int index) {
  if (index < 0 || this.start + index > this.end) {
    throw new IllegalArgumentException("Index out of bound");
  }
  return this.array[this.start + index];
}

public ImmutableSeq<T> slice(int start, int end) {
   return new ImmutableSeq<>(this.array, this.start + start, this.start + end);
}
}
```

To remove aliasing of array, we explicitly copy the array in the factory method `of`

4. **Enabling safe concurrent execution**: correctness ensured (see Concurrent Programming)

13 Nested Class

Nested class: class defined within another containing class

- Used to group logically relevant classes together
- Used to encapsulate information within container class, if implementation of container class becomes too complex (**helper class**)
- Is a field of the containing class

Static Nested Class

Static nested class: can only access static fields and methods of containing class

Inner Class

Inner class (**non-static nested class**): can access all fields and methods of containing class

```
class A {
  private int x;
  private static int y;

  static class C {
    void bar() {
      x = 1; // accessing x from A is not OK since C is static
      y = 1; // accessing y is OK
    }
```

```java
  }

  class B {
    void foo() {
      x = 1; // accessing x from A is OK
      y = 1; // accessing y from A is OK
    }
  }
}
A a = new A();
A.B b = a.new B(); // not new a.B()
A.C c = new A.C();
```

Qualified `this` reference: access instance fields in container class

`<container>.this.<variable>`

Hiding Nested Classes

Declare as `private` $\rightarrow$ inaccessible outside of container class

```java
class A {
  private class B {
    public void buz() { }
  }
  B foo() {
    return new B();
  }
  private static class C {
  }
}
```

```java
// Cannot access A.B or A.C
A.B b; // compilation error
A.C c; // compilation error

// Possible to expose instances of private nested classes
    outside the enclosing class, as long as the type B is
    not used directly
A a = new A();
a.foo(); // return an instance of A.B is OK
A.B b = a.foo(); // still not allowed since A.B is private
```

Local Class

Local class: class declared inside a method (not available outside the method)

- Like a nested class, a local class has access to variables of the enclosing class through qualified `this` reference
- Can access local variables of the enclosing method

```java
void sortNames(List<String> names) {
  class NameComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
      return s1.length() - s2.length();
    }
  }
  names.sort(new NameComparator());
}
```

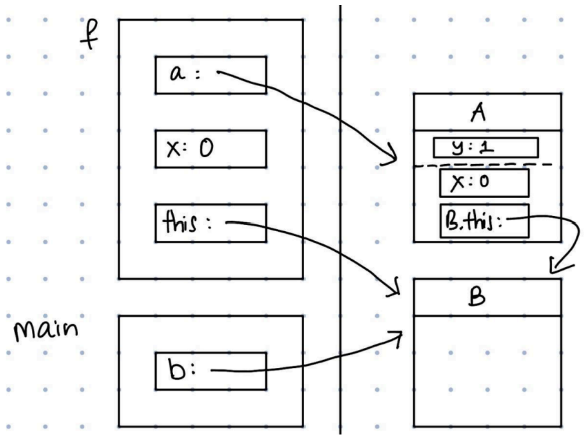
Variable capture: a local class **captures** the local variables

- When a method completes, all local variables are removed from the stack, but an instance of the local class might still exist $\rightarrow$ need to access local variables

- Local class makes a copy of local variables (of the enclosing method) inside itself
- Captured variables must be **declared/effectively final** (cannot be re-assigned)

  Otherwise code does not compile

```java
class B {
  void f() {
    int x = 0; // captured
    class A {
      int y = 0;
      A() {
        this.y = x + 1;
      }
    }
    A a = new A();
  }
}
```



Anonymous Class

Anonymous class: declare local class and instantiate it in a single statement → no name

- "Single-use" class: use only once and never need again
- Format: `new X(arg){ body }`

```java
void sortNames(List<String> names) {
  names.sort(new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
      return s1.length() - s2.length();
    }
  }); // what is the name of this class?
}
```

14 Lambda Expressions

Pure Functions

Pure function:

1. **No side effects**: simply computes and returns the value

   Side effects: print to screen (`System.out.println`), write to files, throw exceptions (e.g. divison by zero), modify fields, modify arguments

2. **Referentially transparent** (**deterministic**): given the same input, the function always produces the same output *every single time*

   If $f(x) = a$, then any uses of $f(x)$ or a can be replaced with its counterpart, with guarantee that the resulting formulas are still equivalent

Functional programming: style of programming where a program is built from pure functions
Functional-style programming: cannot write code consisting of only pure functions, but can write methods that have no side effects & objects that are immutable

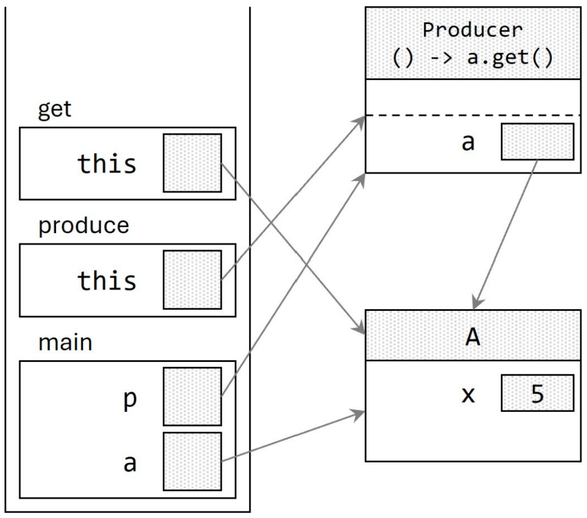
Lambda Expressions

Functional interface: interface with exactly one abstract method

Lambda expression: an anonymous function (declare an anonymous functional interface)

Anonymous function is represented as anonymous class, where type name is compile-time type name & the lambda expression

```
A a = new A(5);
Producer<Integer> p = () -> a.get();
p.produce();
```



Method Reference

Method reference: if a lambda expression does nothing but call an existing method, it is clearer to refer to the existing method by name

1. Static method: `ContainingClass::staticMethodName`
2. Instance method of a particular object: `containingObject::instanceMethodName`
3. Instance method of an arbitrary object of a particular type: `ContainingType::methodName` (first argument is an instance of `ContainingType`)
4. Constructor of a class: `ClassName::new`

```
Box::of          // x -> Box.of(x)
Box::new         // x -> new Box(x)
x::compareTo     // y -> x.compareTo(y)
A::foo           // (x, y) -> x.foo(y) or (x, y) -> A.foo(x,y)
```

Currying

Higher-order function: takes in a single argument, returns another function

```
/**
int add(int x, int y) {
  return x + y;
}
*/
Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> (x + y);
add.transform(1).transform(1);
// add.transform(1) returns the function y -> 1 + y (value of x = 1 is fixed)
Transformer<Integer,Integer> incr = add.transform(1);
```

Currying: translate a general n-ary function (multiple parameters) to a sequence of n unary functions (receive one parameter, return another function)

$$f\colon (X, Y) \to Z \quad \text{to} \quad f\colon X \to (Y \to Z)$$

accepts first argument X, returns a function which accepts second argument Y.
After currying, we have a sequence of **curried functions**.

Closure

Lambda expression is a **closure**: stores

1. the function, and
2. the data from the environment where it is defined (enclosing environment)

Thus a closure allows the function to access those captured variables through the closure's reference to them, even when the function is invoked outside their scope.

```
Point origin = new Point(0, 0);
Transformer<Point, Double> dist = p -> origin.distanceTo(p);
// dist captures the variable origin
```

Note: captured variable must be declared/effectively final

Cross-Barrier State Manipulator

Lambda as a **cross-barrier state manipulator**: change the internals of an object, without exposing them through getter/setter

Box

`map` and `filter` accept a function as a parameter $\to$ allow client to manipulate the data behind abstraction barrier without knowing the internals of the object

```
class Box<T> {
  private T item;
    :

  public <U> Box<U> map(Transformer<? super T, ? extends U> transformer) {
    if (!isPresent()) {
      return empty();
    }
    return Box.ofNullable(transformer.transform(this.item));
  }
    :

  public Box<T> filter(BooleanCondition<? super T> condition) {
    if (!isPresent() || !(condition.test(this.item)) {
      return empty();
    }
    return this;
  }
    :
}
```

Maybe

`Maybe<T>` is an **option type**: a wrapper around a value that is either there or is `null` → write code without mostly not worrying about `NullPointerException` (internalises `null` checks)

Conceptually, `Maybe` is a box: `None` represents having no value (empty box), `Some` is a box with the value inside.

`of`	Creates a `Maybe` containing our value (or `None` if given a `null`)
`filter(BooleanCondition<? super T> predicate)`	If `Some` , check if predicate holds; convert to `None` if predicate is false, otherwise returns this If `None` , propogate the `None`
`map(Transformer<? super T, ? extends U> mapper)`	If `Some` , apply `mapper` on the value If `None` , propogate the `None`
`flatMap(Transformer<? super T, ? extends Maybe<? extends U> > mapper)`	If `Some` , apply `mapper` to the value, which returns a `Maybe` , then flatten the `Maybe` (i.e. no nested `Maybe`) If `None` , propogate the `None`
`orElse(T obj)`	If `Some` , return the value If `None` , return the given `obj`
`orElseGet(Producer<? extends T> obj)`	If `Some` , return the value If `None` , produce it using `Producer`
`ifPresent(Consumer<? super T> obj)`	If `Some` , consume the value If `None` , propogate the `None`

15 Lazy Evaluation

Lambda as Delayed Data

Lazy evaluation: lambda expressions are only evaluated when parameters are supplied → delay execution of the lambda body until it is needed

- Allows the build up of a sequence of complex computations, without actually executing them, until we need to
- Expressions are evaluated on demand when needed (delay computation until we need it)

Memoisation

Memoisation: cache the value of a function, so that future calls to the same function avoid repated (expensive) computations

```java
class Lazy<T> {
  T value;
  boolean evaluated;
  Producer<T> producer;

  public Lazy(Producer<T> producer) {
    evaluated = false;
    value = null;
    this.producer = producer;
  }

  public T get() {
    if (!evaluated) {
      value = producer.produce();
      evaluated = true;
    }
    return value;
  }
}
```

Since the value is evaluated once, the method body is only going to run once

Lazy

`of(T value)`	Initialise a `Lazy` object with the given value
`of(Producer<? extends T> s)`	Takes in a producer (which produces the value when needed)
`get()`	If the value is already available, return that value; otherwise, compute the value and return it.
`map(Transformer<? super T, ? extends U> mapper)`	
`flatMap(Transformer<? super T, ? extends Lazy<? extends U> > mapper)`	
`filter(BooleanCondition<? super T> pred)`	Lazily tests if the value passes the test or not
`combine(Lazy<S> obj, Combiner<? super T, ? super S, ? extends R> combiner)`	Combine two `Lazy` objects (possibly of different types), and return a new Lazy object

16 Infinite List

`InfiniteList` is constructed using a **lazy evaluation**: Each node has a head and tail (lazy producers): store a `Producer`, which generates the head/tail when `produce()` is invoked

```
class InfiniteList<T> {
  private final Producer<T> head;
  private final Producer<InfiniteList<T>> tail;
  // An Infinite List with a head and a tail Infinite List which gives another head and tail
      Infinite List ...

  public static <T> InfiniteList<T> generate(Producer<T> producer) {
    return new InfiniteList<T>(producer,
        () -> InfiniteList.generate(producer));
  }

  public static <T> InfiniteList<T> iterate(T init, Transformer<T, T> next) {
      return new InfiniteList<T>(() -> init,
      () -> InfiniteList.iterate(next.transform(init), next));
  }

  public InfiniteList(Producer<T> head, Producer<InfiniteList<T>> tail) {
    this.head = head;
    this.tail = tail;
  }
```

```java
  public T head() {          // be careful, the method name
    return this.head.produce(); // is the same as the field name
  }

  public InfiniteList<T> tail() { // same here, method name
    return this.tail.produce(); // is the same as field name
  }

  public T get(int n) {
    if (n == 0) {
      return this.head();       // be careful!
    }                           //   use the methods
    return this.tail().get(n - 1); // instead of fields
  }

  public <R> InfiniteList<R> map(Transformer<? super T, ? extends R> mapper) {
    return new InfiniteList<>(
        () -> mapper.transform(this.head()),
        () -> this.tail().map(mapper)); // recursion
  }
}
```

`InfiniteList<T>` behaves like a lazy, infinite linked list:

$$\texttt{head} \rightarrow \texttt{tail} \rightarrow \texttt{tail} \rightarrow \texttt{tail} \rightarrow \cdots$$

`generate(Producer<T> prod)`	Constructs a new `InfiniteList` given a producer. If the producer produces `V`, then the InfiniteList is `[V, V, ...]`
`iterate(T seed, Transformer<? super T, ? extends T> next)`	Construct a new InfiniteList given a seed and the transformer to transform the next seed. `[seed, next(seed), next(next(seed)), ...]`
`head()`	Retrieve the head of the InfiniteList lazily.
`tail()`	Retrieve the tail of the InfiniteList lazily.
`map(Transformer<? super T, ? extends R> func)`	Map each element of this InfiniteList to a new element.
`filter(BooleanCondition<? super T> pred)`	Filter the elements of the InfiniteList. Keep only elements where pred(elem) returns true.
`limit(long n)`	Truncates the InfiniteList to a finite list with at most `n` elements.
`takeWhile(BooleanCondition<? super T> predicate)`	Truncates the InfiniteList as soon as it finds an element that evaluates the condition to false.
`append(InfiniteList<T> list)`	Append `list` to the end of "this" InfiniteList. (This only makes sense if "this" InfiniteList is finite.)
`flatMap(Transformer<? super T, ? extends InfiniteList<R> > mapper)`	Lazily applies a transformer to each element of the InfiniteList, and flattens the resulting nested InfiniteList.
`reduce(U identity, Combiner<U, ? super T, U> accumulator)`	Performs a reduction on the elements of the InfiniteList.
`count()`	Returns the number of elements in the InfiniteList.

17 Streams

Building a Stream

`of(T... values)`	Returns a sequential ordered stream whose elements are the specified values
`generate(Supplier<? extends T> s)`	Returns an infinite sequential unordered stream where each element is generated by the provided Supplier
`iterate(T seed, UnaryOperator<T> f)`	Returns an infinite sequential ordered stream produced by iteratively applying `f` to an initial element `seed`: [`seed`, `f(seed)`, `f(f(seed))`, ...]

Intermediate Operations

Transform a stream into another stream

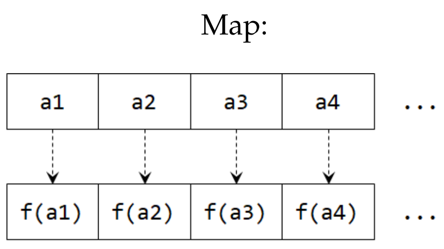
Intermediate operations are lazy and do not cause the stream to be evaluated

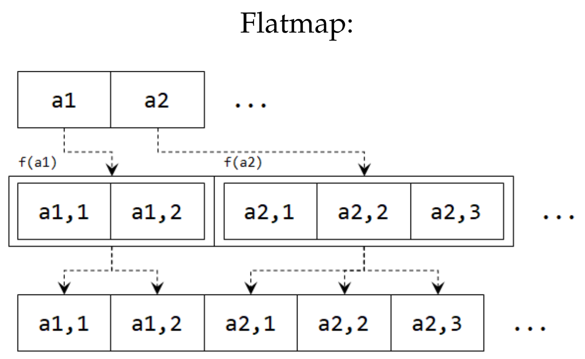
`filter(Predicate<? super T> predicate)`	Returns a stream with elements that satisfy the given predicate
`map(Function<? super T, ? extends R> mapper)`	Returns a stream consisting of the results of applying `mapper` to the elements of this stream
`flatMap(Function<? super T, ? extends Stream<? extends R> > mapper)`	Apply `mapper` to elements, then flatten the resulting streams into a new stream
`sorted()`	Returns a stream with the elements sorted
`distinct()`	Returns a stream with only distinct elements (using `equals` method)
`limit(long n)`	Returns a (finite) stream containing the first `n` elements
`takeWhile(Predicate<? super T> predicate)`	Returns a stream containing elements that satisfy the predicate, terminating when predicate becomes false
`peek(Consumer<? super T> action)`	Returns a stream consisting of the elements of this stream, additionally performing `action` on each element as elements are consumed from the resulting stream.

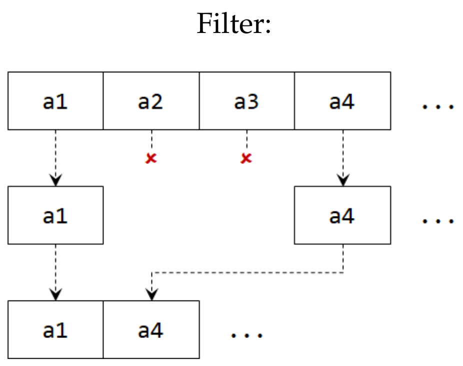
Terminal Operations

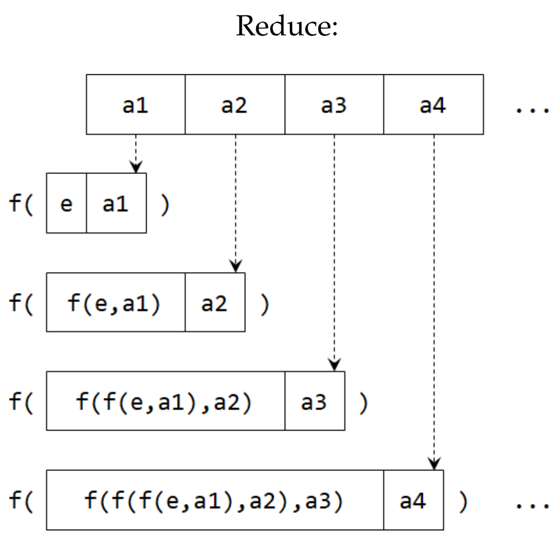
Consume a stream, i.e. trigger the evaluation of the stream

`forEach(Consumer<? super T> action)`	Performs an action for each element of the stream
`reduce(T identity, BinaryOperator<T> accumulator)`	Performs a reduction operation on the elements of stream
`count()`	Returns number of elements in the stream
`allMatch(Predicate<? super T> predicate)`	True if all elements satisfy the predicate
`anyMatch(Predicate<? super T> predicate)`	True if at least one element satisfies the predicate
`noneMatch(Predicate<? super T> predicate)`	True if no elements satisfy the predicate









Consumed Once

A stream can only be **operated once**. We cannot iterate through a stream multiple times. Doing so would lead to `IllegalStateException`. We have to **recreate** the stream if we want to operate on the stream more than once.

```
Stream<Integer> s = Stream.iterate(0, x -> x + 1)
                          .takeWhile(x -> x < 5);
s.forEach(System.out::println);
s.forEach(System.out::println); // Fail!
// We have to recreate the stream
```

18 Monads and Functors

Loggable

After performing an operation, we want to return a string describing the operation (for **logging**).

```
class Loggable<T> {
  private final T value;
  private final String log;

  private Loggable(T value, String log) {
    this.value = value;
    this.log = log;
  }
  public static <T> Loggable<T> of(T value) {
    return new Loggable<>(value, "");
  }

  public <R> Loggable<R> flatMap(
      Transformer<? super T, ? extends Loggable<? extends R>> transformer) {
    Loggable<? extends R> l = transformer.transform(this.value);
    return new Loggable<>(l.value, l.log + this.log);
  }

  public String toString() {
    return "value: " + this.value + ", log: " + this.log;
  }
}
```

`flatMap` takes in a lambda expression `T` $\rightarrow$ `Loggable`, and returns a new `Loggable`

Functors

Functor: class that holds a value (no need to maintain side information)

1. **Identity**: `fct.map(x -> x)` $\equiv$ `fct`
2. **Composition**: `fct.map(x -> f(x)).map(x -> g(x))` $\equiv$ `fct.map(x -> g(f(x)))`

Monads

Monad: class that holds a value & side information (**context**)
For all monads `mn`, functions `f`, `g` which return a monad,

1. **Left identity**: `Monad.of(x).flatMap(x -> f(x))` $\equiv$ `f(x)`

   LHS: wrap the value into the monad, then apply `f` to the value

   RHS: directly apply `f` to the value
2. **Right identity**: `mn.flatMap(x -> Monad.of(x))` $\equiv$ `mn`

   LHS: wrap the value then flatten

   RHS: original monad remains unchanged
3. **Associative**: `mn.flatMap(x -> f(x)).flatMap(y -> g(y))` $\equiv$
   `mn.flatMap(x -> f(x).flatMap(y -> g(y)))`

   LHS: apply `f` to get an intermediate monad, then apply `g` to the value

   RHS: apply one function to `mn`, where the function is `f` then `g`

Monad	Context
`Maybe<T>`	The answer may be missing
`Lazy<T>`	The answer is evaluated when needed and memoised
`Stream<T>`	The answer is one of the item in the stream
`CompletableFuture<T>`	The answer will be available when you need it (no control over when it is actually ready)

Proof of functor (composition):

`m.map(x -> f(x)).map(x -> g(x))`

by our implementation

$\Rightarrow$`m.flatMap(x -> Monad.of(f(x))).flatMap(x -> Monad.of(g(x)))`

by Associative Law

$\Rightarrow$`m.flatMap(x -> Monad.of(f(x)).flatMap(x -> Monad.of(g(x))))`

and by Left Identity Law

$\Rightarrow$`m.flatMap(x -> Monad.of(g(f(x))))`

By our implementation,

$\Rightarrow$`m.map(x -> g(f(x)))`

$\because$ `m.map(x -> f(x)).map(x -> g(x))` $\equiv$ `m.map(x -> g(f(x)))`

Part III

Parallelism & Concurrency

19 Parallel Streams

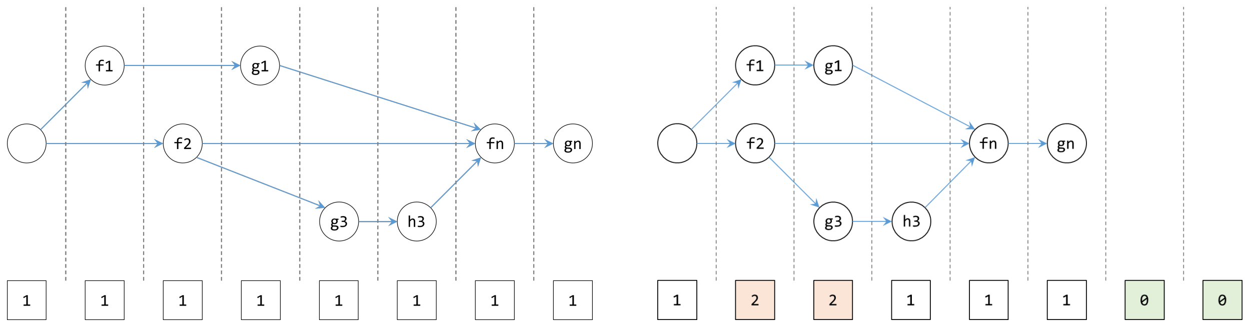
Parallel and Concurrent Programming

Sequential: do things in order on one thread
Concurrent: one thread does multiple things by context switching very quickly (illusion of doing multiple things at the same time)

- Divide computation into subtasks (**threads**)
- **Single-core processor**: executes only one instruction at a time
  OS will switch between processes (**time slicing**) to allow multiple processes to run "at once"

Parallel: actually do multiple things at the same time

- Multiple processors → instructions are distributed across the processors
- Modern computers have more than one core/processor



All parallel programs are concurrent, but not all concurrent programs are parallel

Parallel Stream

`parallel()` : marks a stream to be processed in parallel

- Can be inserted anywhere after the data source & before the terminal operations (since it is a lazy operation)
- How it works
  1. Breaks down stream into subsequences
  2. Runs operations for each subsequence in parallel
  3. Whichever parallel tasks that complete first will output the result to screen first
- `forEach` performs its actions on the units of worked that have already been completed first → the results might be reordered
  `forEachOrdered` follows the encounter order, but some benefits of parallelisation is lost

What Can be Parallelised?

1. **No interference**: stream operations do not modify the source of the stream during execution of terminal operation (e.g. add/remove elements)

   ```
   List<String> list = new ArrayList<>(List.of("Luke", "Leia", "Han"));
   list.stream()
       .peek(name -> {
         if (name.equals("Han")) {
           list.add("Chewie"); // they belong together
         }
       })
       .forEach(i -> {});
   ```

   Results in `ConcurrentModificationException`

2. **Stateless**: result does not depend on any state that might change during execution of the stream

   ```
   Stream.generate(scanner::nextInt)
         .map(i -> i + scanner.nextInt())
         .forEach(System.out::println)
   ```

   generate and map operations below are stateful, since they depend on the state of the standard input. Parallelizing this may lead to incorrect output.

3. **No side effects**: side effects can lead to incorrect results in parallel execution

   ```
   List<Integer> list = new ArrayList<>(
       Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15, 17, 19));
   List<Integer> result = new ArrayList<>();
   list.parallelStream()
       .filter(x -> isPrime(x))
       .forEach(x -> result.add(x));
   // forEach generates a side effect, since it modifies result
   ```

   `ArrayList` is a **non-thread-safe data structure**: If two threads manipulate it at the same time, an incorrect result may result. To resolve this:

   - Use `collect` method
   - Use a thread-safe data structure e.g. `CopyOnWriteArrayList` in `java.util.concurrent`
   - Use `toList` method, which returns a list in the same order as the stream

Parallelisable `reduce`

Parallel reduce: reduce each sub-stream concurrently, then combine the results

`reduce(U e, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`

- `identity` : starting value of each sub-stream
- `accumulator` : accumulator for each sub-stream
- `combiner` : combines the result of each sub-stream's accumulation in the encounter order of elements

A parallelisable operation should produce the same result as sequential operation (using `accumulator` only) when run in parallel (using `accumulator` and `combiner`):

$$((e \oplus a_1) \oplus a_2) \otimes ((e \oplus a_3) \oplus a_4) = (((e \oplus a_1) \oplus a_2) \oplus a_3) \oplus a_4$$

Rules (sufficient conditions) for `reduce` to be **parallelisable**:

1. **Identity**: `e` is the identity

$$e \oplus x = x$$

2. **Associative**: `combiner` and `accumulator` are associative

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$
$$(x \otimes y) \otimes z = x \otimes (y \otimes z)$$

3. **Compatible**: `combiner` and `accumulator` are compatible:
`combiner.apply(x, accumulator.apply(e, y))` $\equiv$ `accumulator.apply(x, y)`

$$x \otimes (e \oplus y) = x \oplus y$$

Performance

Creating a thread incurs some **overhead** $\rightarrow$ the overhead of creating too many threads might outweigh benefits of parallelisation
Unordered streams: created from `generate` or unordered collections (e.g. `Set`)
Ordered streams: created from `of`, `iterate`, ordered collections (e.g. `List`, arrays)

- **Encounter order**: order that elements are added
- **Stable operation**: respect the encounter order (e.g. `distinct`, `sorted`)
- Operations that attempt to preserve encounter order can get expensive, as they need to coordinate between the streams to maintain the order (e.g. `findFirst`, `limit`, `skip`)
- `unordered()`: make parallel operations more efficient, if the original order is not important

20 Asynchronous Programming

Synchronous programming: runs one step at a time

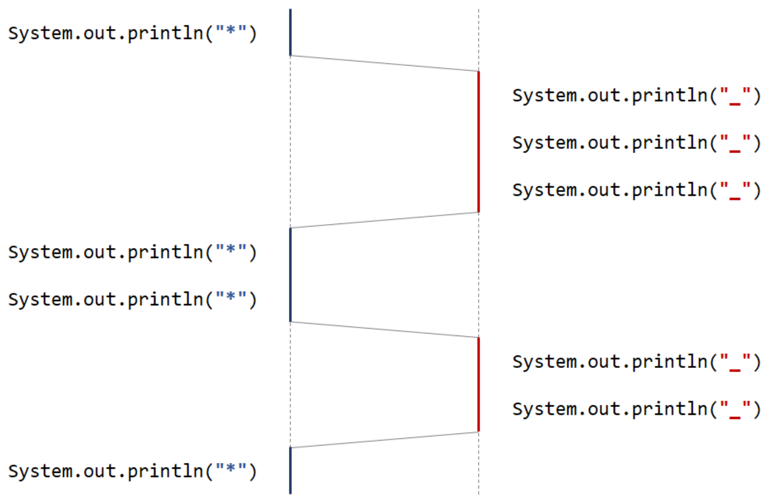
- If a method takes too long to process, the execution of the entire program stalls (the method **blocks** until it returns)
- Inefficient if there are frequent method calls that block for a long period
- The processing can be done in the background instead $\rightarrow$ perform other tasks in the mean time

Threads

Thread: a single flow of execution in a program
Multiple threads $\rightarrow$ run multiple tasks at the same time

`Thread(Runnable task)`	Create a `Thread` instance (encapsulate a function to run in a separate thread)
`start()`	Execute the given lambda expression
`getName()`	Returns the name of the thread
`currentThread()`	Get the reference of the current running thread
`sleep(long millis)`	Pause execution for a given duration
`isAlive()`	Check if a thread is still running

`Runnable` is a functional interface with a method `run()` that takes in no parameter, returns `void`
Every time the same program is run, different **interleaving** of executions → no control over which thread is executed first, when it switches to another



Limitations of `Thread`

1. **Coordination**: no mechanism to specify execution order and dependencies among threads
2. **Exceptions**: if an exception is thrown and not caught within the thread (e.g. via `try`-`catch`), the thread terminates silently & other threads continue running normally
3. **Overhead**: threads are single use (cannot run more than once) → creating new threads takes up some resources → more efficient to reuse threads to run multiple tasks
4. **Sharing of data**: race condition (multiple threads attempt to read from and/or write to the same data structure → might override each other)

```java
// Sum up values
int n = 1_000;
int[] arr = new int[n];
// int sum = 0; doesn't work since not effectively final
int[] sum = new int[] { 0 };

for (int i = 0; i < n; i++) {
  arr[i] = 1;
}

Thread t1 = new Thread(() -> {
  for (int i = 0; i < n/2; i++) {
    sum[0] = += arr[i]
  }
});
```

```
Thread t2 = new Thread(() -> {
  for (int i = n/2; i < n; i++) {
    sum[0] += arr[i];
  }
});

t1.start();
t2.start();
// the output is not 1000!
```

The first thread reads the value as 100, takes out 100. At the same time, the second thread reads the value as 100, takes out 100. Then the first thread evaluates $100 + 1 = 101$, puts back 101; the second thread also does the same. Hence the two threads override each other.

CompletableFuture

Asynchronous programming: run other things while a method continues to run

`CompletableFuture` : a monad that helps us parallelise our code in asynchronous fashion

- Encapsulates a value (which is the value that you want to compute, or the result of a computation)
- Context: this value has been produced, or may not be produced yet (eventually it will produce the value, some time in the future)

A `CompletableFuture` **completes** when the operation (lambda expression) that you pass in has finished execution and returns a value.

Non-blocking operation: does not cause the executing thread to pause and wait for the operation to complete $\rightarrow$ instead, the thread can continue with other tasks while it is processed in the background (program will continue to execute)

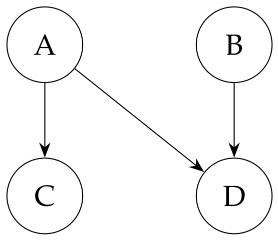
Blocking operation: program stops and waits for `this` to complete, before it continues execution

`completedFuture(U value)`	Create a new `CompletableFuture` that is already completed with the given value
`runAsync(Runnable runnable)`	Create a new `CompletableFuture` that is asynchronously completed by a task running in the ForkJoin-Pool.commonPool() after it runs the given action
`supplyAsync(Supplier<U> supplier)`	Create a new `CompletableFuture` that is asynchronously completed by a task running in the ForkJoin-Pool.commonPool() with the value obtained by calling the given Supplier.
`allOf`	Create a new `CompletableFuture` which is completed only when all the input completes
`anyOf`	Create a new `CompletableFuture` which is completed only when any one of the input completes
`thenApply(Function<T, U> fn)`	Analogous to `Stream::map` (run the given lambda expression in the same thread as the caller)
`thenCompose(Function<T, CompletionStage<U> > op)`	Analogous to `Stream::flatMap` `op` is logically executed after `this` completes Operation is non-blocking
`thenCombine(CompletableFuture<U> other, BiFunction<T, U, V> op)`	Analogous to `Stream::combine` `op` is logically executed after `this` and `other` complete Operation is non-blocking
`thenRun(Runnable action)`	Executes `action` after `this` is completed
`runAfterBoth(CompletableFuture<?> other, Runnable action)`	Executes `action` after `this` <u>and</u> `other` are completed
`runAfterEither(CompletableFuture<?> other, Runnable action)`	Executes `action` after `this` <u>or</u> `other` is completed
`get()`	Get the result
`join()`	Get the result Operation is **blocking**

Asynchronous version: `thenApplyAsync`, `thenComposeAsync`, `thenCombineAsync`, `thenRunAsync`, `runAfterBothAsync`, `runAfterEitherAsync` cause the given lambda expression to run in a different thread (thus more concurrency)

```
// Runs on two different threads asynchronously
CompletableFuture<Integer> ith = CompletableFuture.supplyAsync(
  () -> findIthPrime(i));
CompletableFuture<Integer> jth = CompletableFuture.supplyAsync(
  () -> findIthPrime(j));
// Combines the results of the two threads
CompletableFuture<Integer> diff = ith.thenCombine(jth, (x, y) -> x - y);
// Waits for all threads to complete computation
diff.join();
```

Dependency diagram among tasks: figure out what gets printed by looking at all possible orders of execution



Exception handling: `handle((value, exception)-> exception == null ? value : default value)`

- Suppose a computation in a thread throws an exception
- The exception is stored and passed down the chain of calls, until `join()` is called
- `join()` throws `CompletionException` (which contains the original exception), and whoever calls `join()` will be responsible for handling this exception

```
// This throws CompletionException
CompletableFuture.<Integer>supplyAsync(() -> null)
                 .thenApply(x -> x + 1)
                 .join();
// This handles the exception and uses 0 as the default value
cf.thenApply(x -> x + 1)
  .handle((t, e) -> (e == null) ? t : 0)
  .join();
```

21 Fork and Join

Analogy: **thread** is a worker (ready to work), **task** is something that you ask that person to do for you

Thread Pool

Too many threads created due to recursive structure → too much overhead → reuse threads to do multiple tasks

```
// A trivial thread pool with a single thread
Queue<Runnable> tasks = new LinkedList<>();
Thread scheduler = new Thread(() -> {
    while (true) {
      if (!tasks.isEmpty()) {
        Runnable r = tasks.remove();
        r.run();
      }
    }
  });

for (int i = 0; i < 100; i++) {
  int count = i;
  tasks.add(() -> System.out.println(count));
}

scheduler.start();
```

Thread pool: (i) a collection of threads, each waiting for a task to execute, and (ii) a collection of tasks (in a **shared queue**) - idle thread picks up a task from the shared queue to execute

Fork and Join

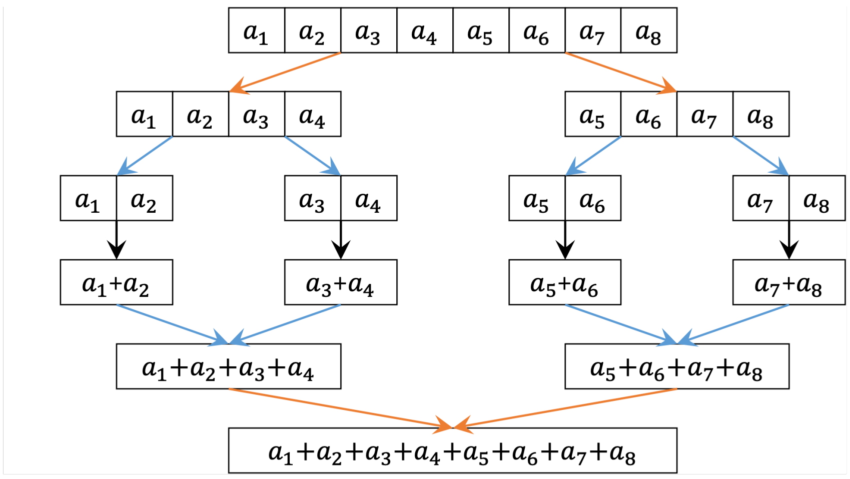
Fork-join model: a parallel divide-and-conquer model of computation

- breaking up the problem into identical problems but with a smaller size (**fork**)

- solve the smaller version of the problem recursively
- combine the results (**join**)

`RecursiveTask<T>`

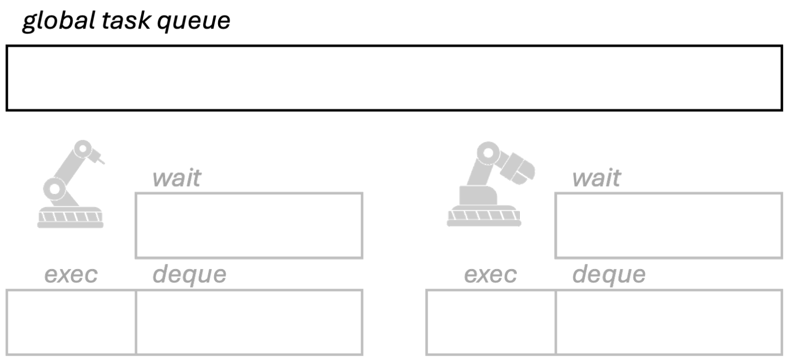
`fork()`	Divide a problem (submits a smaller version of the task for execution) • This does not execute the spawned task directly (it only creates task) • This does not block the current task (code will continue to execute) • This does not block the current thread
`join()`	Retrieve the result (waits for the smaller tasks to complete and return) • If the result is not yet available, the current task may be suspended. Then the joined task may be executed. • This does not necessarily block the current thread ("worker" continues to work, on different tasks)
`compute()`	Define the task's logic • If the task is small enough, solve directly • If not, split it into subtasks, fork them to run in parallel, then join their results.


a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
a_1+a_2
a_3+a_4
a_5+a_6
a_7+a_8
$a_1+a_2+a_3+a_4$
$a_5+a_6+a_7+a_8$
$a_1+a_2+a_3+a_4+a_5+a_6+a_7+a_8$

```java
// Recursively sums up the content of an array
class Summer extends RecursiveTask<Integer> {
  private static final int FORK_THRESHOLD = 2;
  private int low;
  private int high;
  private int[] array;

  public Summer(int low, int high, int[] array) {
    this.low = low;
    this.high = high;
    this.array = array;
  }

  @Override
  protected Integer compute() {
    // stop splitting into subtask if array is already small.
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) {
        sum += array[i];
      }
      return sum;
    }

    int middle = (low + high) / 2;
    Summer left = new Summer(low, middle, array);
    Summer right = new Summer(middle, high, array);
    left.fork();
    return right.compute() + left.join();
  }
}

Summer task = new Summer(0, array.length, array);
int sum = task.compute();
```

`ForkJoinPool`

Uses a fixed number of threads to parallelise the computation by automatic thread management



Main thread:

- `fork()` : adds the forked task to the back of the global task queue
- `join()` : puts the current task in the waiting area, waits for the result from the joined task to be available

Worker thread:

- Idle thread: checks its deque of tasks
  - If not empty → pick up a task at the head of its deque → execute it (using `compute()`)
  - If empty → pick up a task from the tail of deque of another thread (**work stealing**) → execute it
- `fork()` : adds the forked task to the head of local task deque
- `join()` : puts the current task in the waiting area

    - If the joined task is in the deque, execute the joined task by calling `compute()`
    - Otherwise, wait for the result from the joined task to be available - may execute other tasks in the meantime

Palindrome ordering: `join()` in reverse order of `fork()`

- Since the most recently forked task is likely to be executed next, we should `join()` the most recent `fork()` task first
- If not will need to do some pops and pushes to get to the subtask we want $\rightarrow$ less efficient

```
left.fork(); // >------------+
right.fork(); // >---------+ | should have
return right.join() // <--+ | no crossing
    + left.join(); // <-----+
```