CS2030

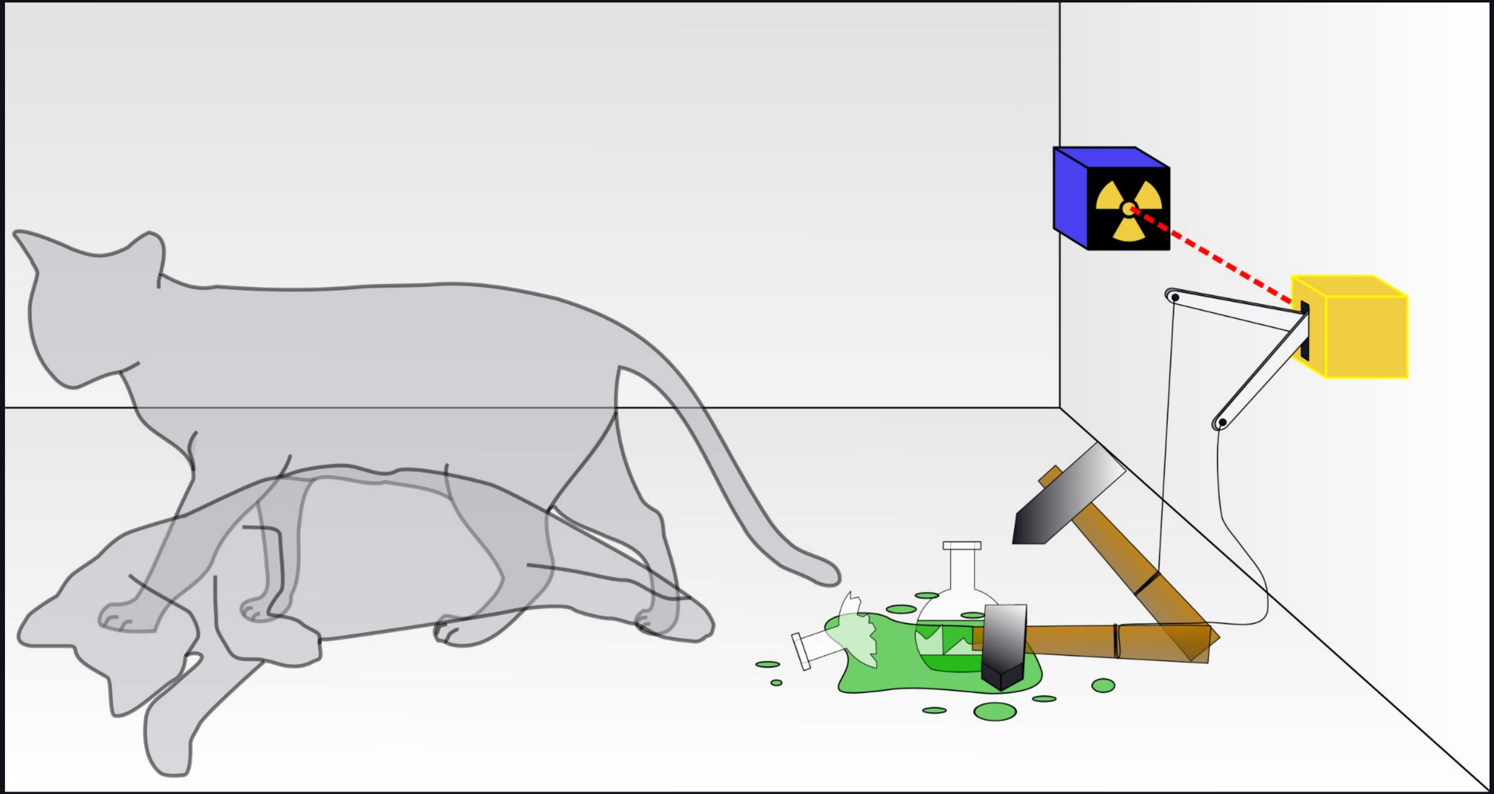
Lab 6

AY25/26 Sem 1, Week 11

Fadhil Peer Mohamed <f_p_m@u.nus.edu>
Abner Then <abner.then@u.nus.edu>

Schrödinger's Cat

Schrödinger's Cat



Iteration 1

Iteration 1

- Create an instance of Random
  - Generate a double
  - If it’s larger than 0.5, print “Cat lives”
  - Otherwise, print “Cat dies”

```java
import java.util.Random;

class Main {
    public static void main(String[] args) {
        Random RNG = new Random();

        double liveChance = RNG.nextDouble();

        if (liveChance > 0.5) {
            System.out.println("Cat lives");
        } else {
            System.out.println("Cat dies");
        }
    }
}
```

Iteration 2

Iteration 2

- Your boss complains that they’re unable to tell if your code is about flipping a coin or about Schrodinger’s cat
- They demand immediate changes...

Iteration 2 – OOP

- Encapsulation and Abstraction
- Object-oriented model
- Immutability

```java
class Cat {
    private final String name;

    Cat(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat: " + name;
    }
}
```

Iteration 2 - OOP

- **Object-oriented model**
  - Group related data and behaviour
  - Box encapsulates a Cat instance and a Random instance
  - Box also contains experiment() - related behaviour

```java
class Box {
    private final Cat cat;
    private static final Random RNG = new Random();

    Box(Cat cat) {
        this.cat = cat;
    }

    public Box experiment() {
        double liveChance = RNG.nextDouble();
        if (liveChance > 0.5) {
            return this;
        }
        return new Box(null);
    }
    <...>
}
```

Iteration 2 - OOP

- **Object-oriented model**
  - Group related data and behaviour
- **Usage of private helper methods**
  - handle null within Box
  - avoid exposure of null to the outside world
    - avoid risk of NullPointerException

```java
class Box {
    <...>

    private boolean isEmpty() {
        return cat == null;
    }

    public String toString() {
        if (!isEmpty()) {
            return String.format("Box<%s>", cat);
        }
        return "Box<Dead Cat>";
    }
}
```

Iteration 2 – OOP

- **Object-oriented model**
  - Group related data and behaviour
- **More intuitive code**
  - Tell-don’t-ask

```
class Main {
    public static void main(String[] args) {
        Cat cat = new Cat("Mittens");
        Box box = new Box(cat);
        box = box.experiment();
        System.out.println(box);
    }
}
```

Iteration 3

Iteration 3

- One of your coworkers is bellyaching about your Box only being able to hold Cats

- You try to hold them off by writing new Box classes that encapsulate whatever they want, whenever they think of a new one

- Your sanity frays...

Iteration 3

- **Box to Box<T>**
  - Box can now encapsulate any class
  - Avoid writing redundant classes containing identical logic

```
class Box<T> {
    private final T thing;
    private static final Random RNG = new Random();

    Box(T thing) {
        this.thing = thing;
    }

    public Box experiment() {
        double liveChance = rng.nextDouble();
        if (liveChance > 0.5) {
            return this;
        }
        return new Box(null);
    }
    <...>
```

Iteration 3

- **Records**
    - Succinct declaration of immutable classes
    - Provide
        - default constructor
        - getters
        - equals()
        - hashCode()
        - toString()
    - Can also contain user-defined methods

```java
class Cat { // old
    private final String name;

    Cat(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat: " + name;
    }
}
-------------------------------------------------------
record Cat(String name) { // new
    @Override
    public String toString() {
        return "Cat: " + name;
    }
}
```

Iteration 3

- **Pipelining**
  - chain instructions for compactness

```java
class Main { // old
    public static void main(String[] args) {
        Cat cat = new Cat("Mittens");
        Box<Cat> box = new Box<Cat>(cat);
        box = box.experiment();
        System.out.println(box);
    }
}

-----------------------------------------------------

class Main { // new
    public static void main(String[] args) {
        Box<Cat> box = new Box<Cat>(new Cat("Mittens"))
            .experiment();
        System.out.println(box);
    }
}
```

Iteration 4

Iteration 4

- No good deed goes unpunished
- Another coworker is now demanding the ability to write new experimental routines
- They want these routines to also have the ability to change the type of object stored within the Box
- One must imagine Sisyphus happy...

Iteration 4

- **Problem / Coworker demands**
  - They must be able to write the experimental logic themselves (**implementation**)

  - Box must be capable of taking in that experimental logic, and applying it to whatever it encapsulates (**behaviour**)

- **Solution**
  - Write a **generic interface** that describes a **behaviour**
  - Let others **implement** it for their own purposes

Iteration 4

- **Vastly improved flexibility**
  - **Box** can now hold anything
  - **Box** also supports incoming logic
    - convert whatever is inside **Box** to whatever the logic specifies
    - If the **Box** is empty, the **Box stays empty**
    - You can't apply logic to something that doesn't exist

```
interface Converter<T, R> {
    R convert(T thing);
}

class Box<T> {
    <...>
    public <R> Box<R> convert(Converter<T, R> conv) {
        if (!isEmpty()) {
            return Box.<R>of(conv.convert(thing));
        }
        return Box.<R>empty();
    }
    <...>
}
```

Iteration 4

- **Implementing Converter<T, R>**
  - Added flexibility in specifying custom logic using interfaces
  - Transfer default experiment to an implementation of Converter<Cat, Cat>

```java
class Main {
    public static void main(String[] args) {
        Random RNG = new Random();
        double liveChance = RNG.nextDouble();

        Converter<Cat, Cat> experiment = ???

        Box<Cat> box = Box.<Cat>of(new Cat("Mittens"))
            .convert(experiment);
        System.out.println(box);
    }
}
```

Iteration 4

- **Implementing Converter<T, R>**
  - **Option 1:** Declare a traditional class **within** the main method (local class)
  - Very verbose...

```
<...>
Random RNG = new Random();
double liveChance = RNG.nextDouble();

class Experiment implements Converter<Cat, Cat> {
    private final double liveChance;

    Experiment(double liveChance) {
        this.liveChance = liveChance;
    }

    @Override
    public Cat convert(Cat thing) {
        return liveChance > 0.5 ? thing : null;
    }
}

Converter<Cat, Cat> experiment = new Experiment(liveChance);
<...>
```

Iteration 4

- **Implementing Converter<T, R>**
    - **Option 2:** Declare an ***anonymous local class***
    - Relies on **variable capture**
    - Still rather verbose...

```java
<...>
Random RNG = new Random();
double liveChance = RNG.nextDouble();

Converter<Cat, Cat> experiment = new Converter<Cat, Cat>() {
    @Override
    public Cat convert(Cat thing) {
        return liveChance > 0.5 ? thing : null;
    }
};
<...>
```

Iteration 4

- **Implementing Converter<T, R>**
    - **Option 3:** Use a ***lambda expression***
    - Compact implementation

```
<...>
Random RNG = new Random();
double liveChance = RNG.nextDouble();

Converter<Cat, Cat> experiment = x -> liveChance > 0.5
    ? x
    : null;
<...>
```

Iteration 4

- **Implementing Converter<T, R>**
  - **Option 3:** Use a ***lambda expression***
  - Lambda expressions can have multiple lines
  - Tradeoff between compactness and readability

```
<...>
Random RNG = new Random();
double liveChance = RNG.nextDouble();

Converter<Cat, Cat> experiment = x -> {
    if (liveChance > 0.5) {
        return x;
    }
    return null;
};
<...>
```

Iteration 4

Q: What can we implement using a lambda expression?

A: Any interface with a **Single Abstract Method** **(SAM)**. Lambda expressions specify one set of input(s) and output(s), which can correspond to only one implementation of behaviour

Iteration 4

- Your boss now complains that they want to create an empty Box without passing null as a parameter.

- Well...

Iteration 4

- **Factory methods**
  - Static methods for object creation without new keyword
  - Often used for instantiation of generic objects

```
class Box<T> {
    private final T thing;

    private Box(T thing) {
        this.thing = thing;
    }

    public static <T> Box<T> of(T thing) {
        if (thing == null) {
            return Box.<T>empty();
        }
        return new Box<T>(thing);
    }

    public static <T> Box<T> empty() {
        return new Box<T>(null);
    }
    <...>
}
```

Iteration 5

Iteration 5

- Your boss sees your brand new implementation of Converters and flexibility in experimental logic
- They figure out an angle to criticise it. Somehow.
- Apparently, passing null in as a possible output is demoralising some (unspecified) coworker

Iteration 5

- **Addition of Tester**
  - allows definition of logic used to clear the Box if a certain condition is not met

```
interface Tester<T> {
    boolean test(T thing);
}

class Box<T> {
    <...>
    public Box<T> test(Tester<T> tester) {
        return !isEmpty() && tester.test(thing)
            ? this
            : Box.<T>empty();
    }
    <...>
}
```

Iteration 5

- Added flexibility

```
class Main {
    public static void main(String[] args) {
        Random RNG = new Random();
        double liveChance = RNG.nextDouble();

        Tester<Cat> experiment = x -> liveChance > 0.5;

         Box<Cat> box = Box.<Cat>of(new Cat("Mittens"))
            .test(experiment) ;
        System.out.println(box);
    }
}
```

Iteration 6

Iteration 6

- A visiting scholar sees your code and notices that you’ve reinvented the wheel.
- Apparently the Java API already has near identical constructs.

Iteration 6

- Switch to functional interfaces from java.util.Function

```
class Box<T> { // old
    <...>
    public <R> Box<R> convert(Converter<T, R> converter) {
        return !isEmpty() ? Box.<R>of(converter.convert(thing)) : Box.<R>empty();
    }

    public Box<T> test(Tester<T> tester) {
        return !isEmpty() && tester.test(thing) ? this : Box.<T>empty();
    }
    <...>
}
```

Iteration 6

- Switch to functional interfaces from java.util.Function

```java
class Box<T> { // new
    <...>
    public <R> Box<R> map(Function<T, R> converter) {
        return !isEmpty() ? Box.<R>of(converter.apply(thing)) : Box.<R>empty();
    }

    public Box<T> filter(Predicate<T> tester) {
        return !isEmpty() && tester.test(thing) ? this : Box.<T>empty();
    }
    <...>
}
```

Iteration 6

- Yet another coworker was experimenting with **Box** and found a strange bug

- They were running the same experiment on **Box**es containing objects of different types

- Since all classes (besides primitives) in Java **extend Object**, they wrote a **Predicate<Object>** describing experimental logic

- The logic seems to work for all these **Object**s as long as they're outside the **Box**...

Iteration 6

```
jshell> Predicate<Object> pred = x -> new Random().nextDouble() > 0.5;
pred ==> $Lambda/0x000076153400ba40@7d9d1a19
jshell> Cat cat = new Cat("Mittens")
cat ==> Cat: Mittens

jshell> Dog dog = new Dog("Clifford")
dog ==> Dog: Clifford

jshell> pred.test(cat)
$14 ==> true

jshell> pred.test(dog)
$15 ==> false

jshell> Box.<Cat>of(cat).filter(pred)
|  Error:
|  incompatible types: java.util.function.Predicate<java.lang.Object> cannot be converted to
java.util.function.Predicate<Cat>
|  Box.<Cat>of(cat).filter(pred)
|                          ^--^
```

Iteration 6

- **Problem: Java generics are invariant**
  - Even though a **Cat** is-a **Object**, a **Predicate<Cat>** is-NOT-a **Predicate<Object>** and vice-versa
  - **Analogy:**
    - Say you and your parent stay in different homes,
    - You are still your parent’s child
    - Your home is definitely not the child of your parent’s home
  - But **Predicate<Object>** should be accepted by filter...
    - As **Cat** is-a **Object**, **Predicate<Object>**s are able to **consume** **Cat** instances and test them

Iteration 6

- The same coworker was trying to convert a **Box<Cat>** to a **Box<Number>** using the map method.

- They wrote a **Function<Object, Integer>**, and tried to pass it into **Box::map**.

- It didn’t work.
    - Function<Cat, Integer> didn’t work either
    - Function<Object, Number> also didn’t work

Iteration 6

- **Problem: Java generics are invariant**
  - Even if **Cat** is-an **Object** and **Integer** is-a **Number**,
    - a Function<Object, Number> **is-NOT-a** Function<Cat, Number>
    - a Function<Cat, Integer> **is-NOT-a** Function<Cat, Number>
    - a Function<Object, Integer> **is-NOT-a** Function<Cat, Number>

  - **But these Functions should be accepted!**
  - **A function that consumes supertypes of T can certainly consume a T**
    - If an **Object** is consumed, a **Cat** can be consumed.
  - **A function that produces a subtype of R certainly produces R**
    - If a **Number** is promised, returning an **Integer** should fine.

Iteration 6

- **Producer Extends, Consumer Super**
- Function<T, R> -> <u>**consumes**</u> T, <u>**produces**</u> R
- Predicate<T> -> <u>**consumes**</u> T, <u>**produces**</u> boolean
- **Therefore,**

```java
class Box<T> { // without wildcards
    <...>
    public <R> Box<R> map(Function<T, R> converter) {
        return !isEmpty() ? Box.<R>of(converter.apply(thing)) : Box.<R>empty();
    }

    public Box<T> filter(Predicate<T> tester) {
        return !isEmpty() && tester.test(thing) ? this : Box.<T>empty();
    }
    <...>
}
```

Iteration 6

- **Producer Extends, Consumer Super**
- Function<T, R> -> **consumes** T, **produces** R
- Predicate<T> -> **consumes** T, **produces** boolean
- **Therefore,**

```
class Box<T> { // with wildcards
    <...>
    public <R> Box<R> map(Function<? super T, ? extends R> converter) {
        return !isEmpty() ? Box.<R>of(converter.apply(thing)) : Box.<R>empty();
    }

    public Box<T> filter(Predicate<? super T> tester) {
        return !isEmpty() && tester.test(thing) ? this : Box.<T>empty();
    }
    <...>
}
```

Iteration 7

Iteration 7

- Your coworker runs experiments on two separate **Box** instances
- They want to evaluate if both **Cat**s survived
- You tell them to just print both **Box** instances to the command line
- Somehow this won’t do
  - They want to somehow put both **Cat**s into the same **Box**, if both survive.

```
class Main {
    public static void main(String[] args) {
        Random RNG = new Random();

        Box<Cat> boxOne = Box.<Cat>of(new Cat("Mittens"))
            .filter(x -> RNG.nextDouble() > 0.5);

        Box<Cat> boxTwo = Box.<Cat>of(new Cat("Garfield"))
            .filter(x -> RNG.nextDouble() > 0.5);

    }
}
```

Iteration 7

- **Attempt #1:** Nested maps
- Works, but nested **Box**es result
- What if we wanted to do this with 3 **Box** instances?

```
jshell> Random RNG = new Random()
RNG ==> java.util.Random@7cd62f43
jshell> Predicate<Object> expt = x -> RNG.nextDouble() > 0.5
expt ==> $Lambda/0x00007f06dc00b410@4883b407
jshell> Box<Cat> boxOne = Box.of(new Cat("Mittens")).filter(expt);
boxOne ==> Box<Cat: Mittens>
jshell> Box<Cat> boxTwo = Box.<Cat>of(new Cat("Garfield")).filter(expt);
boxTwo ==> Box<Cat: Garfield>
jshell> boxOne.map(x ->
   ...>          boxTwo.map(y -> y.toString() + ", " + x.toString()));
$... ==> Box<Box<Cat: Garfield, Cat: Mittens>>
```

Iteration 7

- **flatMap**
  - The beating heart of any Monad / Computational Context
  - **Exercise:** Justify the usage of wildcards in this flatMap signature

```java
class Box<T> {
    <...>
    public <R> Box<R> flatMap(Function<? super T, ? extends Box<? extends R>> flatMapper) {
        if (!isEmpty()) {
            Box<? extends R> flatMapResult = flatMapper.apply(thing);
            return Box.<R>of(flatMapResult.thing);
        }
        return Box.<R>empty();
    }
    <...>
}
```

Iteration 7

- **flatMap**
  - The beating heart of any Monad / Computational Context
  - Allows combination of innards of two Monads without breaking abstraction
  - flatMap operations are what define a Monad
  - Any map operation can be expressed as a flatMap operation

```
Box<Cat> boxOne = Box.<Cat>of(new Cat("Mittens"))
    .filter(x -> RNG.nextDouble() > 0.5);

Box<Cat> boxTwo = Box.<Cat>of(new Cat("Garfield"))
    .filter(x -> RNG.nextDouble() > 0.5);

Box<String> combinedBox = boxOne.flatMap(x -> boxTwo.map(y -> y.toString() + ", " + x.toString()));
```

Iteration 8

Iteration 8

- The reason behind your boss’s seemingly irrational demands in previous iteration has finally been revealed!

- It appears that they’ve been diagnosed with incurable, terminal **null allergy**

- You’ve got to limit the usage of null as far as possible in **Box**

Iteration 8

- Right now, **Box<T>** either contains an instance of **T**, or a null
- All logic in **Box<T>** checks to see if an instance of **T** is contained before picking the appropriate operation to perform.
- **One** method signature, **two** implementations
- Sound familiar?

Iteration 8

- **Strategy**
  - Convert **Box** to an **interface**.
  - Separate the logic for each case (non-null / null) into two separate **anonymous implementations** returned by **static factory methods** of and empty.

```java
interface Box<T> {
    public <R> Box<R> map(Function<? super T, ? extends R> mapper);

    public <R> Box<R> flatMap(Function<? super T, ? extends Box<? extends R>> flatMapper);

    public Box<T> filter(Predicate<? super T> tester);

    public static <T> Box<T> of(T thing) {<...>}

    public static <T> Box<T> empty() {<...>}
}
```

Iteration 8

```java
public static <T> Box<T> of(T thing) {
    return thing == null ? Box.<T>empty() : new Box<T>() {
        public <R> Box<R> map(Function<? super T, ? extends R> mapper) {
            return Box.<R>of(mapper.apply(thing));
        }

        public <R> Box<R> flatMap(Function<? super T, ? extends Box<? extends R>> flatMapper) {
            Box<? extends R> result = flatMapper.apply(thing);
            return result.<R>map(x -> x);
        }

        public Box<T> filter(Predicate<? super T> tester) {
            return tester.test(thing) ? this : Box.<T>empty();
        }

        public String toString() {
            return String.format("Box<%s>", thing.toString());
        }
    };
}
```

Iteration 8

```java
public static <T> Box<T> empty() {
    return new Box<T>() {
        public <R> Box<R> map(Function<? super T, ? extends R> mapper) {
            return Box.<R>empty();
        }

        public <R> Box<R> flatMap(Function<? super T, ? extends Box<? extends R>> flatMapper) {
            return Box.<R>empty();
        }

        public Box<T> filter(Predicate<? super T> tester) {
            return Box.<T>empty();
        }

        public String toString() {
            return "Box.empty";
        }
    };
}
```

Finale

- **Achievement:**
  - Construction of an Optional/Maybe-like monad in 8 iterations

- **Challenge:**
  - **Implement the following methods in Box<T>**
    - ifPresent
    - ifPresentOrElse
    - or
    - orElse
    - orElseGet

Lab 6

Try<T>

- You will modify the Try interface for exception handling.
- Try_skel.java has been provided, containing code from Lecture 8
- Try_skel.java has a cyclic dependency
- Begin by breaking this cyclic dependency using anonymous inner classes

Lab 6

Deadline: **6 Nov (Thurs) 2359**