EXPLORING ADVANCED OBJECT-ORIENTED CONCEPTS: INHERITANCE, POLYMORPHISM, AND DESIGN PATTERNS

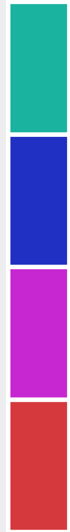
A trial lecture for a Doctor of Philosophy (Dr. Philos.) degree

Trial Lecture

Majid Rouhani
Date: May 2, 2024

NTNU

AGENDA



- Introduction
- Inheritance
- Polymorphism
- Design Patterns

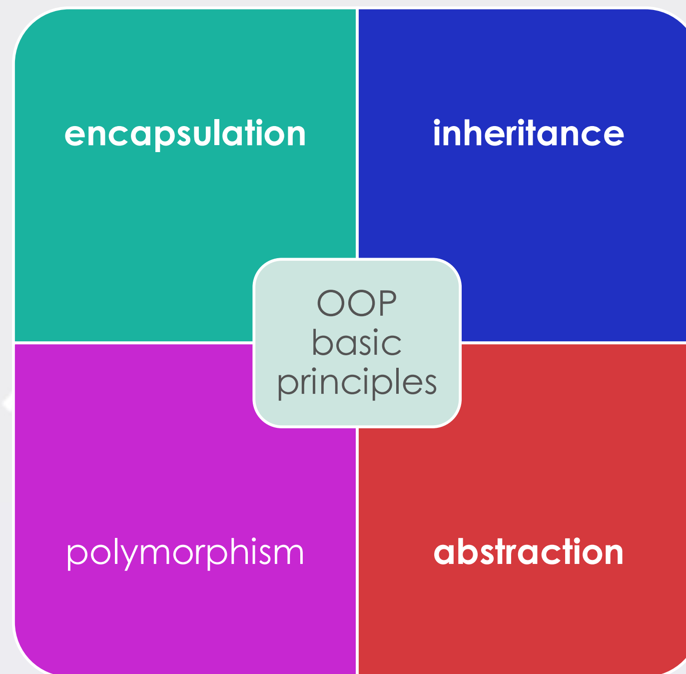
Trial Lecture

NTNU

THE OBJECTIVE

Providing an in-depth exploration and understanding of inheritance, **polymorphism** and design patterns through practical demonstrations.

BASIC PRINCIPLES



By embracing OOP principles developers can create **maintainable and scalable** applications [8].

THE CASE: COFFEE MAKER MACHINE

Select type of beverage

Prepare selected beverage

Hot water

Waterlogic (Push button: Designed by: Freepik)

Designed by: Freepik

Collection of orders

Designed by: Freepik

TYPE THEORY[4]

- Acts like a set of **rules** that guides computers to handle **data** correctly, preventing **errors**.

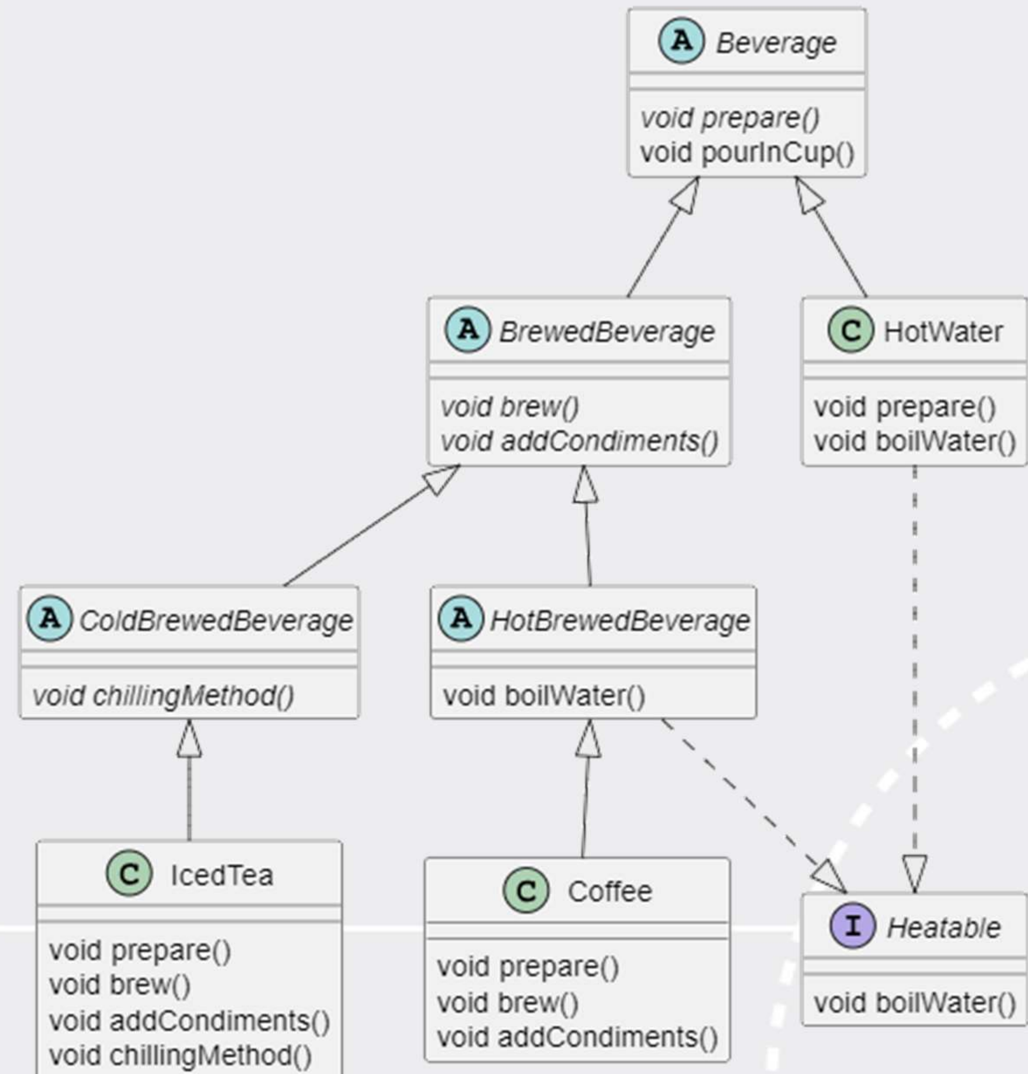
- Type theory is important in programming because it helps ensure
  – high-quality,
  – reliable,
  – and efficient code.

- It is **a broad and extensive field** that has been studied and developed for many years.

Designed by Freepik

INHERITANCE

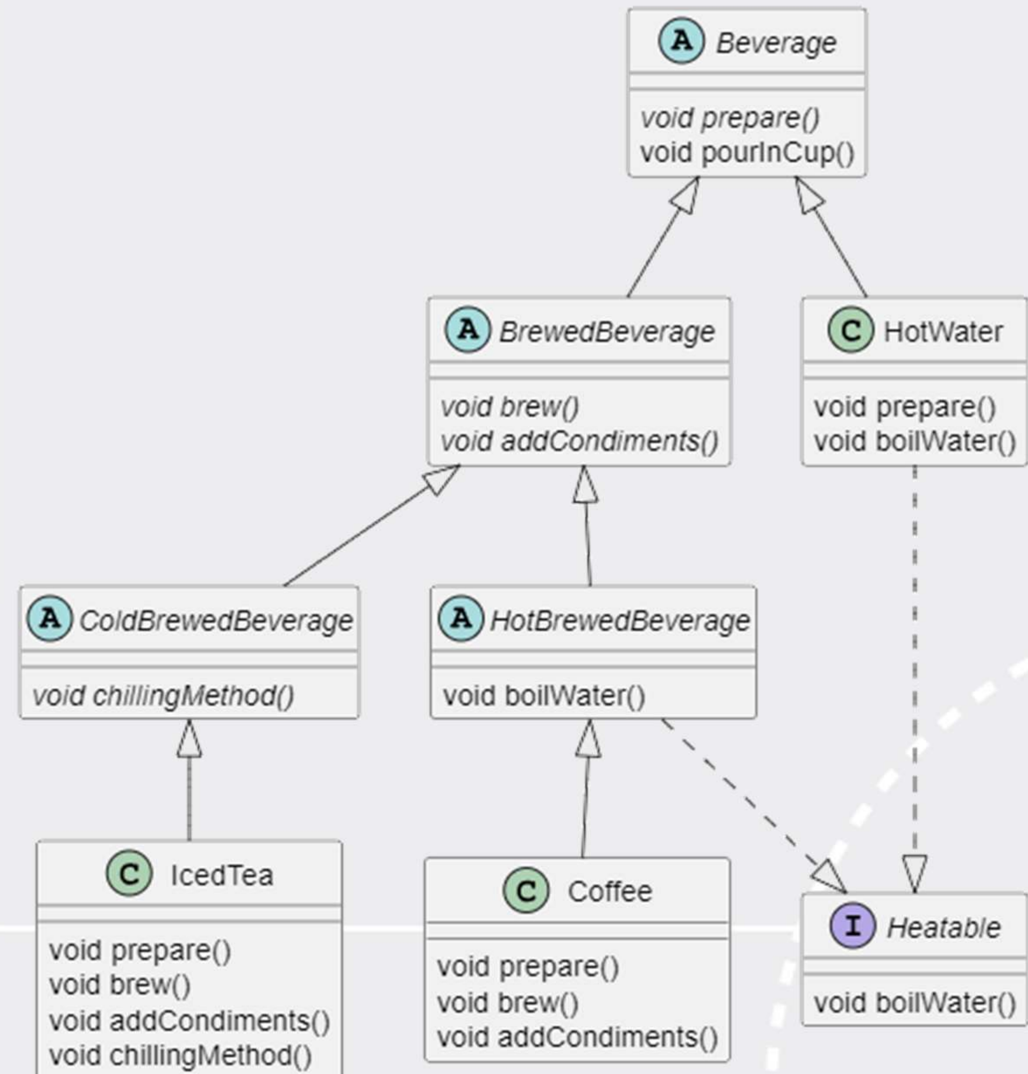
- Allows a **new class (subclass)** to inherit properties and behaviors from an **existing class (superclass)**.
- Subclasses can extend or **override** the functionality of the superclass.
- This promotes **code reusability** and helps in creating a hierarchical relationship between classes.



Trial Lecture

TYPE THEORY FACILITATING ABSTRACTION

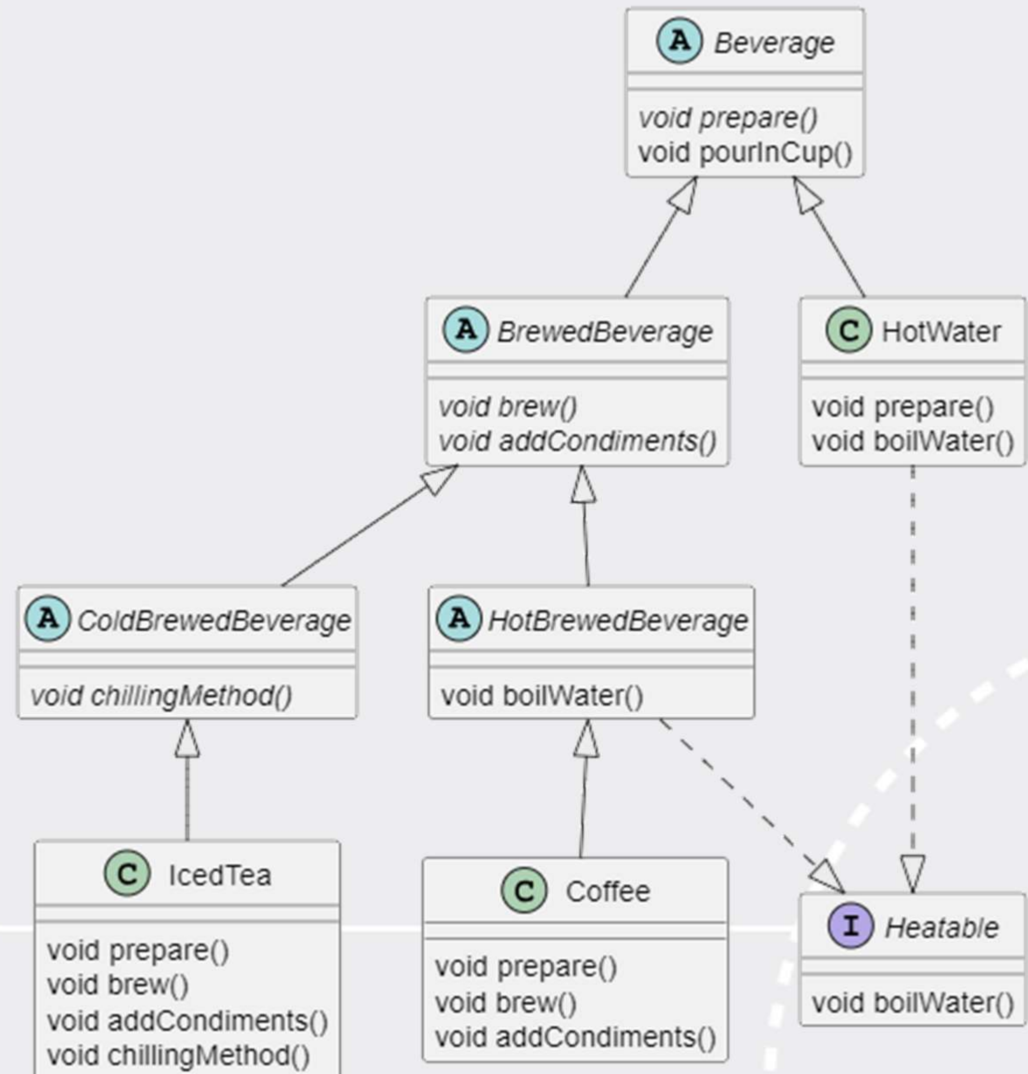
- Provide **common** functionality to derived classes.
- Can't be instantiated directly.



Trial Lecture

TYPE THEORY FACILITATING POLYMORPHISM

Different types can be used **interchangeably** if they adhere to the same interface or type class



ABSTRACT CLASS EXAMPLE

Definition

```
abstract class Beverage {abstract void prepare();}
…

class Tea extends HotBrewedBeverage {…}
…

Collection<BrewedBeverage> getBrewedBeverageOrders() {
    return List.of(new Coffee(), new Tea(), new IcedTea());
}

void prepareBrewedBeverages(Collection<? extends BrewedBeverage> brewedBeverages) {
    brewedBeverages.forEach(brewedBeverage -> {
          brewedBeverage.prepare();
      });
}
```

Usage

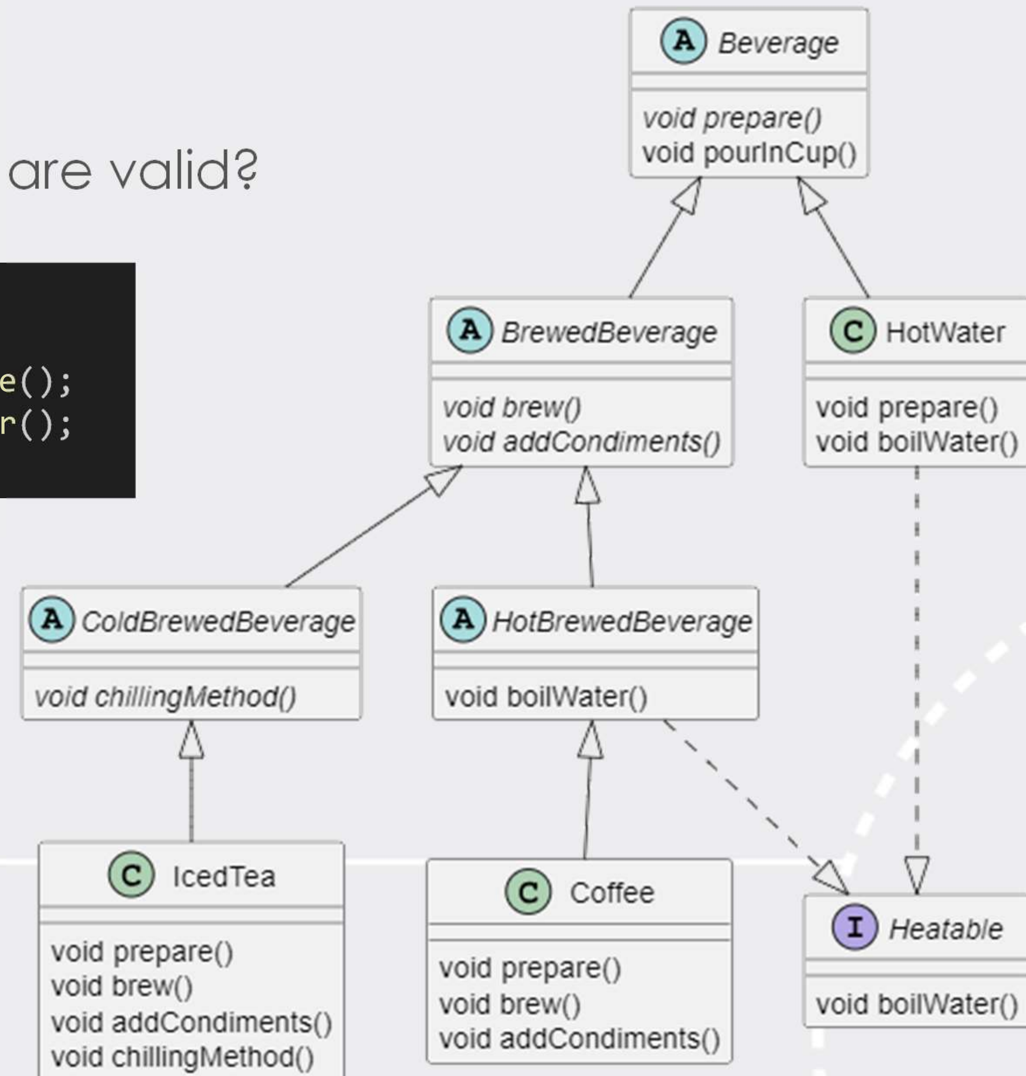
```
Collection<BrewedBeverage> onlyBrewedBeverages = getBrewedBeverageOrders();

prepareBrewedBeverages(onlyBrewedBeverages);
```

QUESTION

Which of the class instantiations are valid?

```
1:        Coffee coffee = new Coffee();
2:        Beverage beverage = new Beverage();
3:        HotWater hotWater = new HotWater();
```



Trial Lecture

POLYMORPHISM

Select type of beverage

Prepare selected beverage

Hot water

Polymorphism

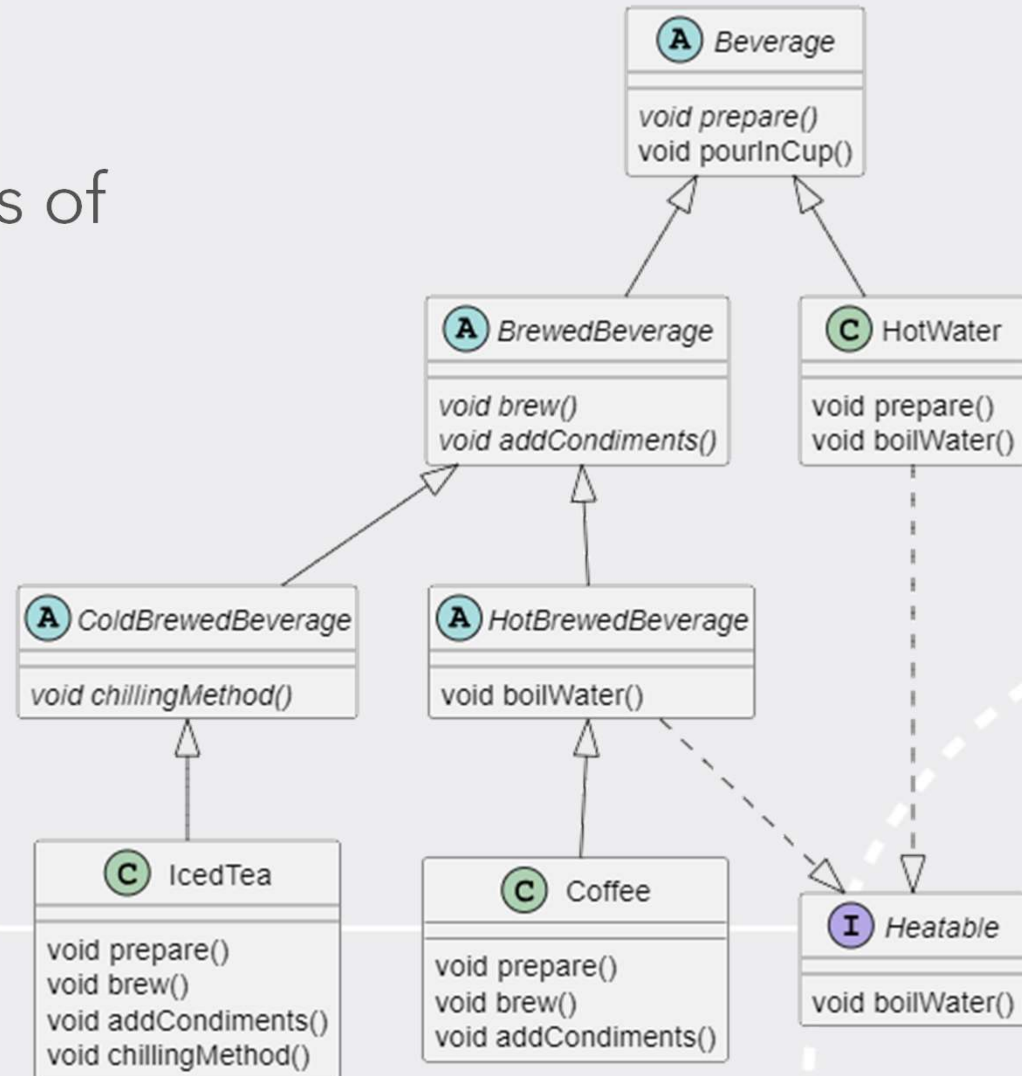
Waterlogic (Push button: Designed by: Freepik)

Designed by: Freepik

POLYMORPHISM

Several different types of polymorphism:

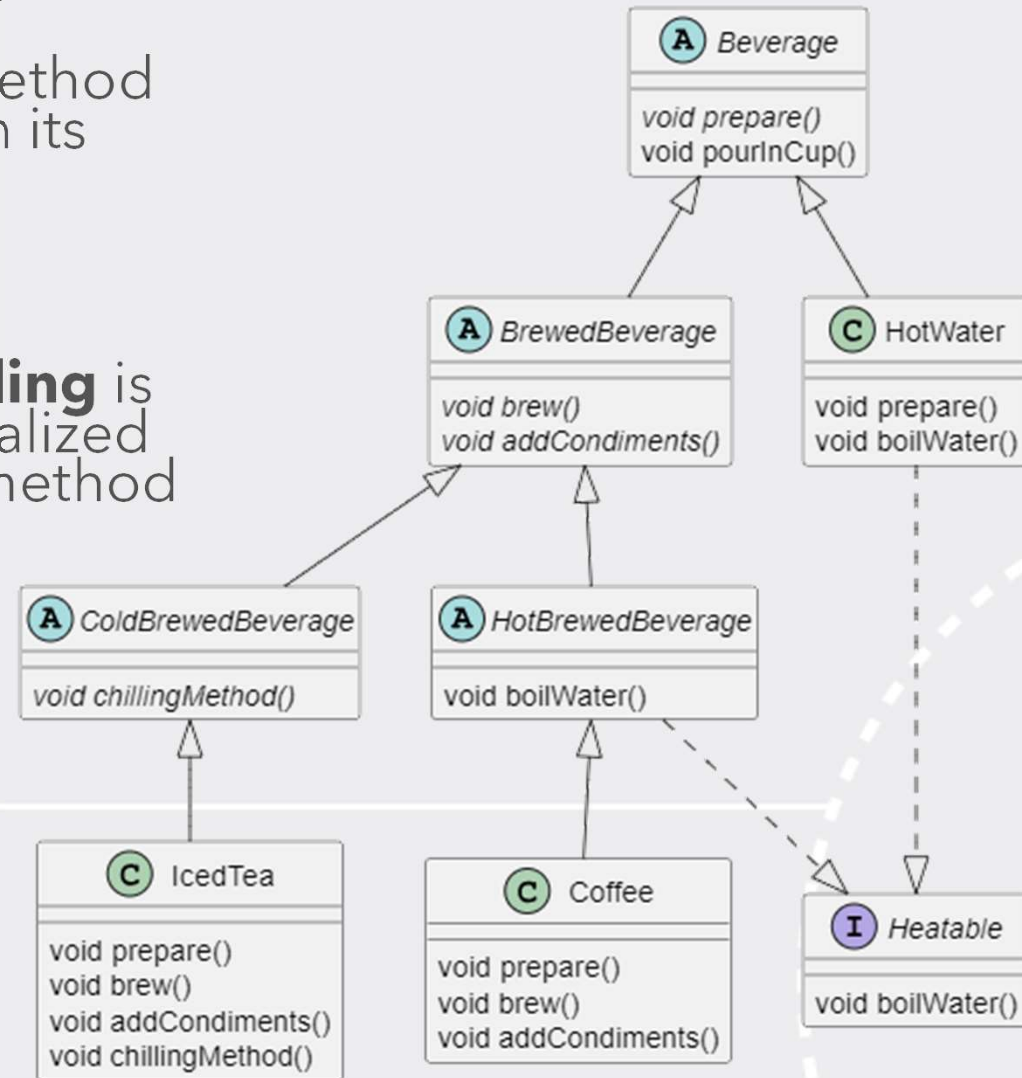
- Sub-type
- Override
- Parametric
- …



OVERRIDING

- Occurs when **a subclass provides a specific implementation** of a method that is already defined in its superclass.

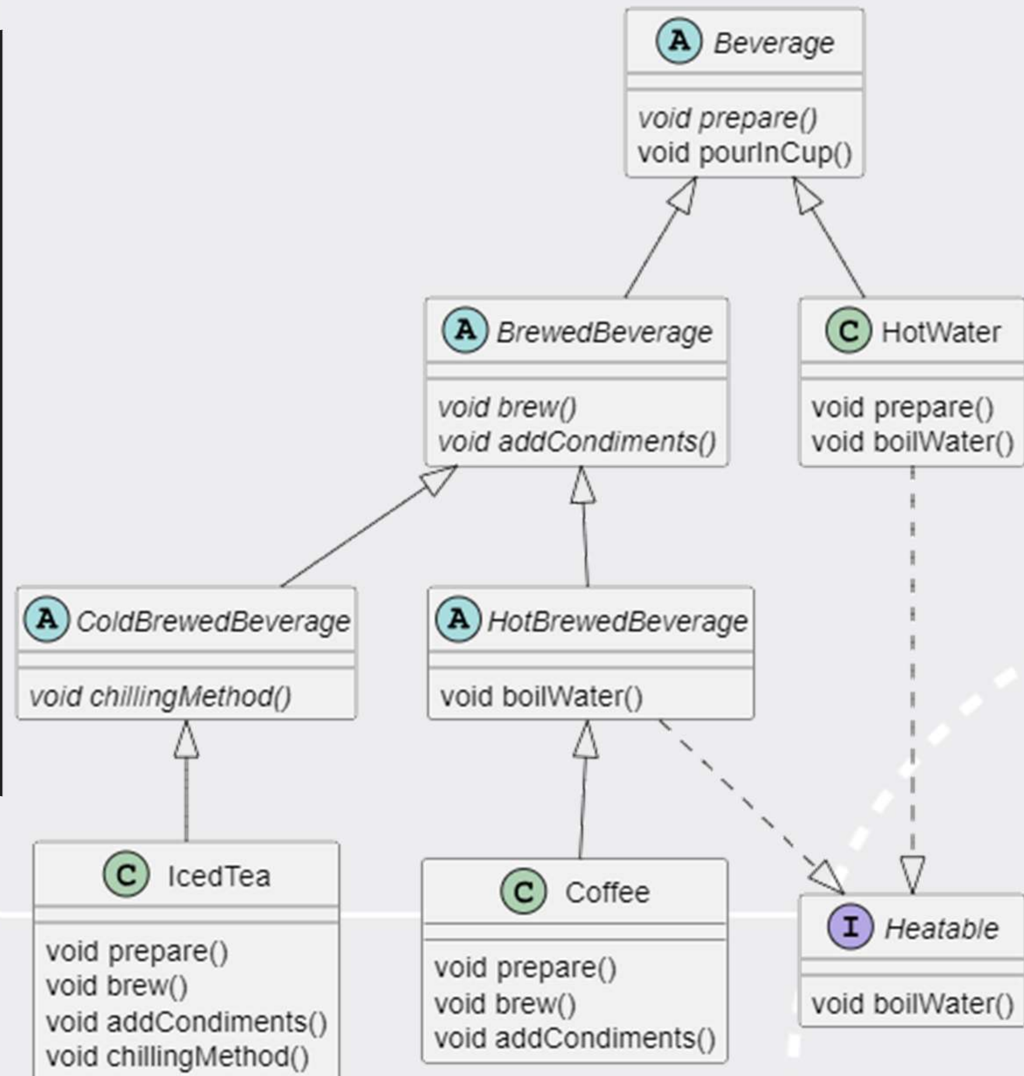
- The **purpose of overriding** is to provide a more specialized implementation of the method in the subclass.



CODE EXAMPLE

```
class Coffee extends HotBrewedBeverage {
    public void prepare() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
…
}

class IcedTea extends ColdBrewedBeverage {
    public void prepare() {
        brew();
        chillingMethod();
        pourInCup();
        addCondiments();
    }
…
}
```



COFFEE MAKER MACHINE

Waterlogic (Push button: Designed by: Freepik)

It can make different types of beverages

- Americano
- Espresso
- IcedTea
- ...

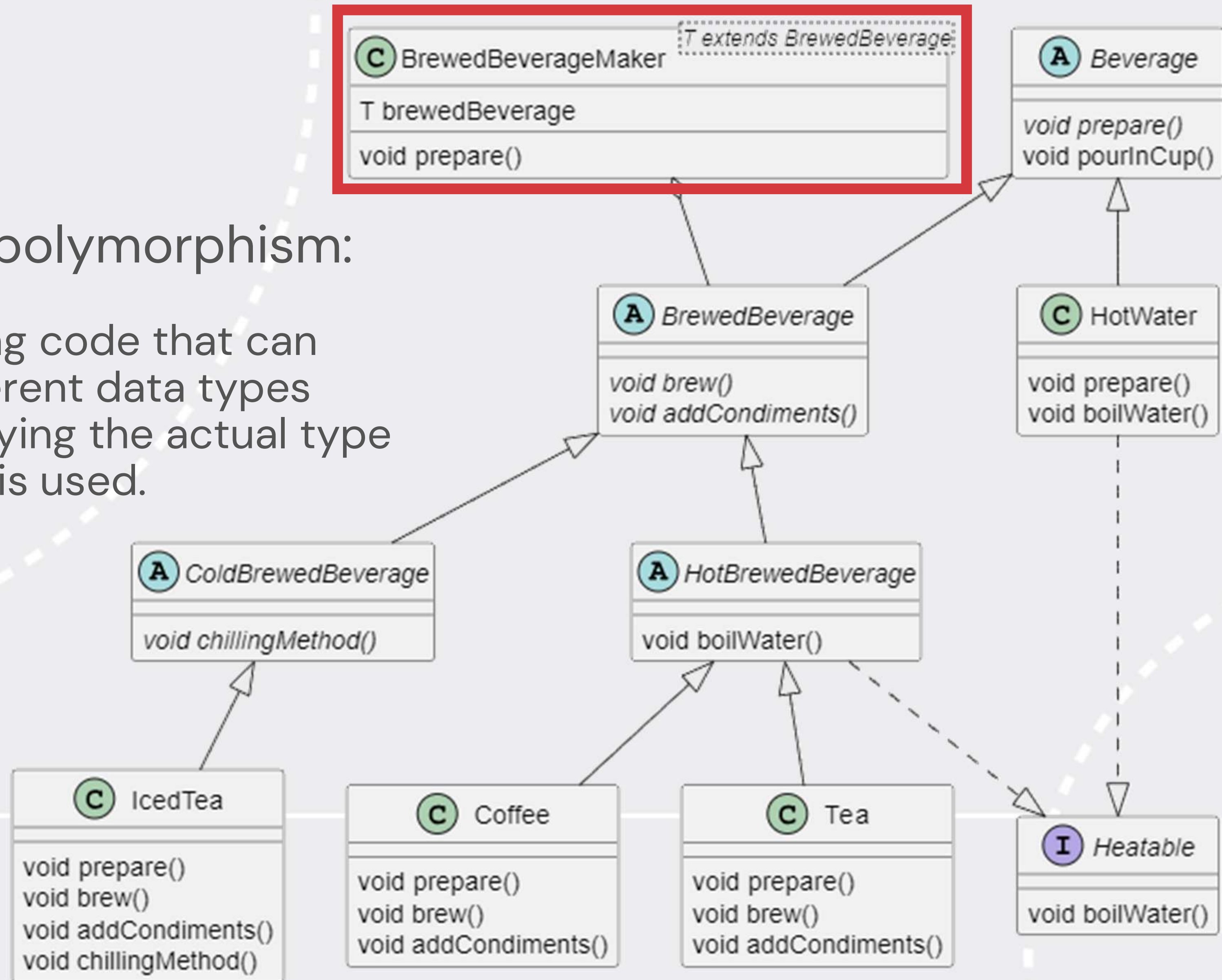
Polymorphism

One machine makes different beverage types > It is generic

GENERIC TYPES: PARAMETERIZED TYPES

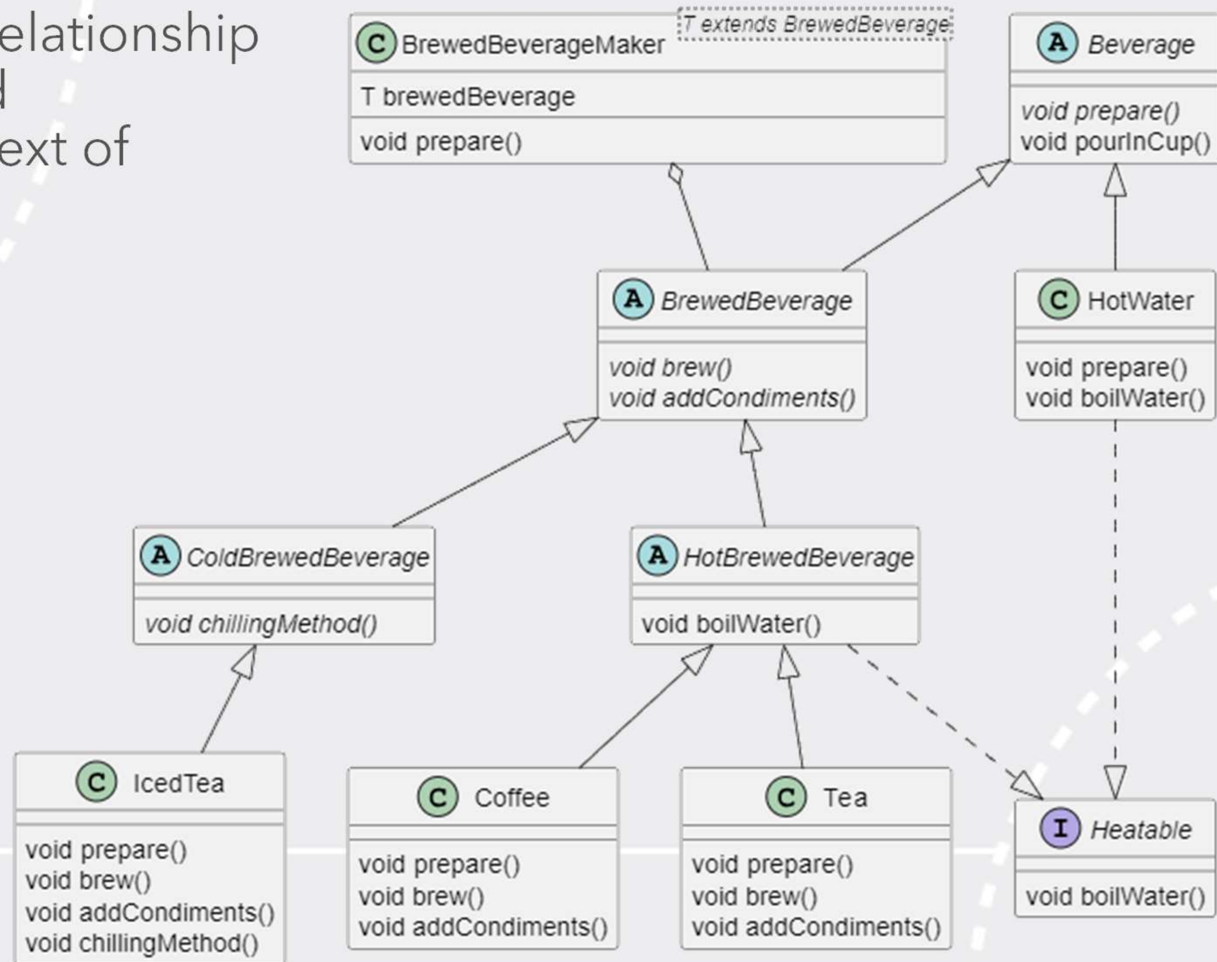
Parametric polymorphism:

Allow for writing code that can work with different data types without specifying the actual type until the code is used.



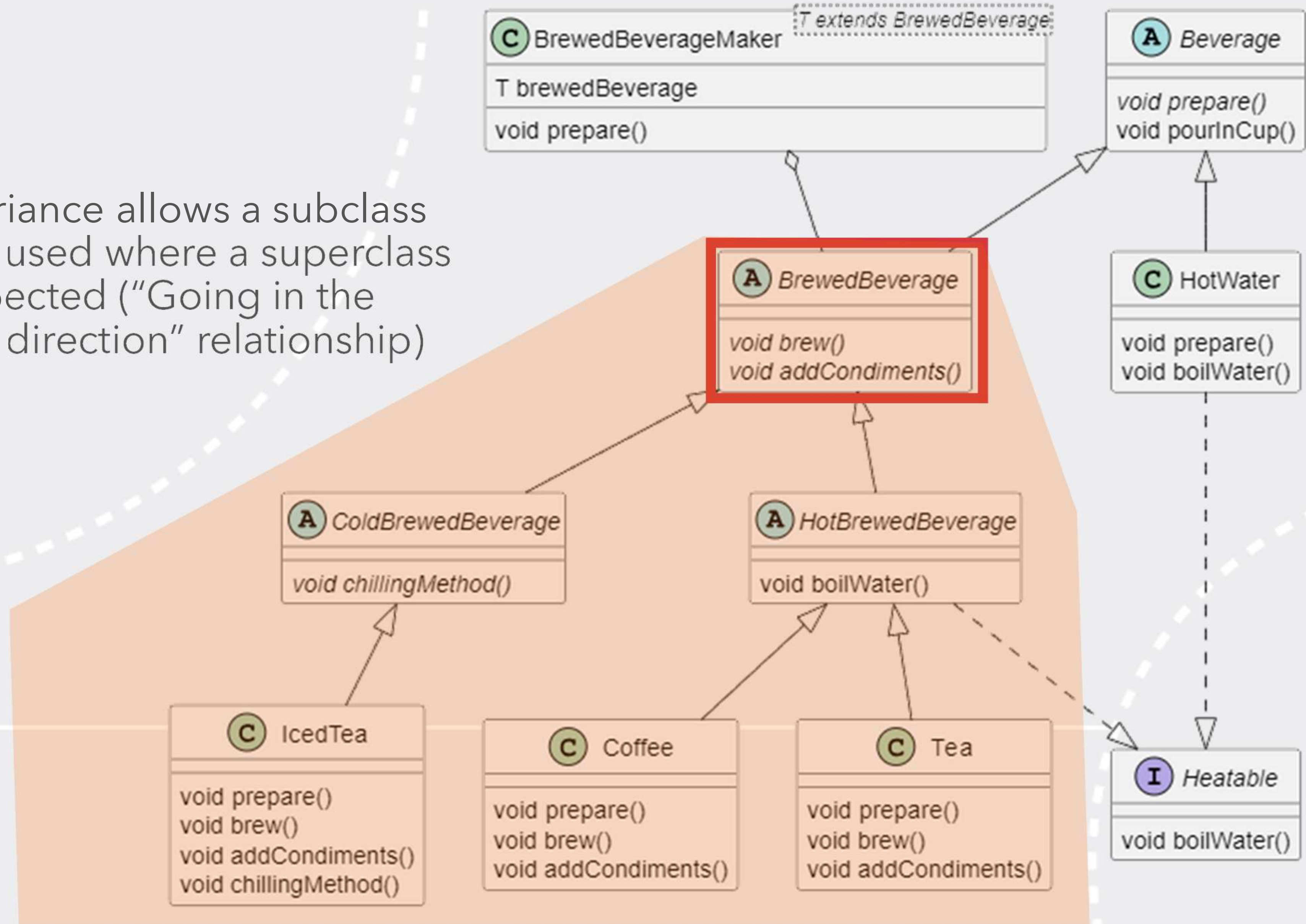
THE RELATIONSHIP BETWEEN SUBTYPES & SUPERTYPES: VARIANCE

- Variance is about the relationship between subtypes and supertypes in the context of generics
- Three types
  - Covariance
  - Contravariance
  - Invariance



COVARIANCE (UPPER BOUND)

Covariance allows a subclass to be used where a superclass is expected ("Going in the same direction" relationship)



COVARIANCE: EXAMPLE

```
Collection<? extends BrewedBeverage> getAnyBrewedBeverageOrders() {

        return List.of(new Coffee(), new Tea(), new IcedTea());

}

void prepareBrewedBeverages(Collection<? extends BrewedBeverage> brewedBeverages) {

          brewedBeverages.forEach(brewedBeverage -> { brewedBeverage.prepare();});

}


Collection<? extends BrewedBeverage> anyBrewedBeverages = getAnyBrewedBeverageOrders();

prepareBrewedBeverages(anyBrewedBeverages);
```

QUESTION

Will this code compile?

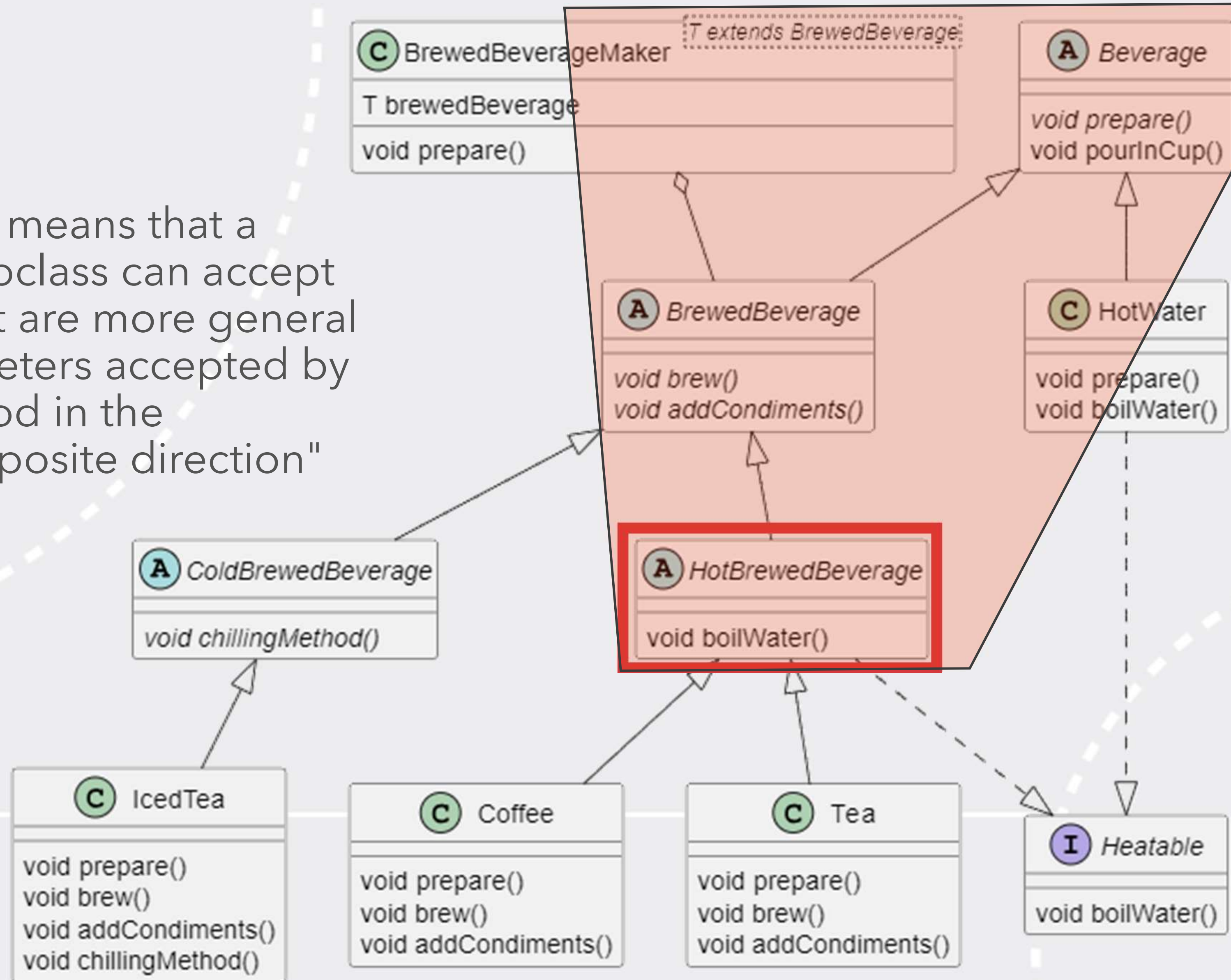
```
void prepareBrewedBeverages(Collection<? extends BrewedBeverage> brewedBeverages) {
    brewedBeverages.forEach(brewedBeverage -> {brewedBeverage.prepare();});
}


Collection<? extends Beverage> anyBeverages = getAnyBeverageOrders();


prepareBrewedBeverages(anyBeverages);
```

CONTRAVARIANCE (LOWER BOUND)

Contravariance means that a method in a subclass can accept parameters that are more general than the parameters accepted by the same method in the superclass ("opposite direction" relationship).



Trial Lecture

CONTRAVARIANCE: EXAMPLE

```
Collection<? extends Beverage> getAnyBeverageOrders() {
    return List.of(new Coffee(), new Tea(), new IcedTea(), new HotWater());
}

Collection<HotBrewedBeverage> getHotBrewedBeverageOrders() {
    return List.of(new Coffee(), new Tea());
}
```

```
void prepareHotBrewedBeverage(Collection<? super HotBrewedBeverage> hotBrewedBeverages) {

  hotBrewedBeverages.forEach(hotBrewedBeverage -> {

            if (hotBrewedBeverage instanceof HotBrewedBeverage)

                ((HotBrewedBeverage) hotBrewedBeverage).prepare();

        });

}

Collection<HotBrewedBeverage> hotBrewedBeverages = getHotBrewedBeverageOrders();

prepareHotBrewedBeverage(hotBrewedBeverages);
```

QUESTION

Which of the calls to prepareHotBrewedBeverage() are valid?

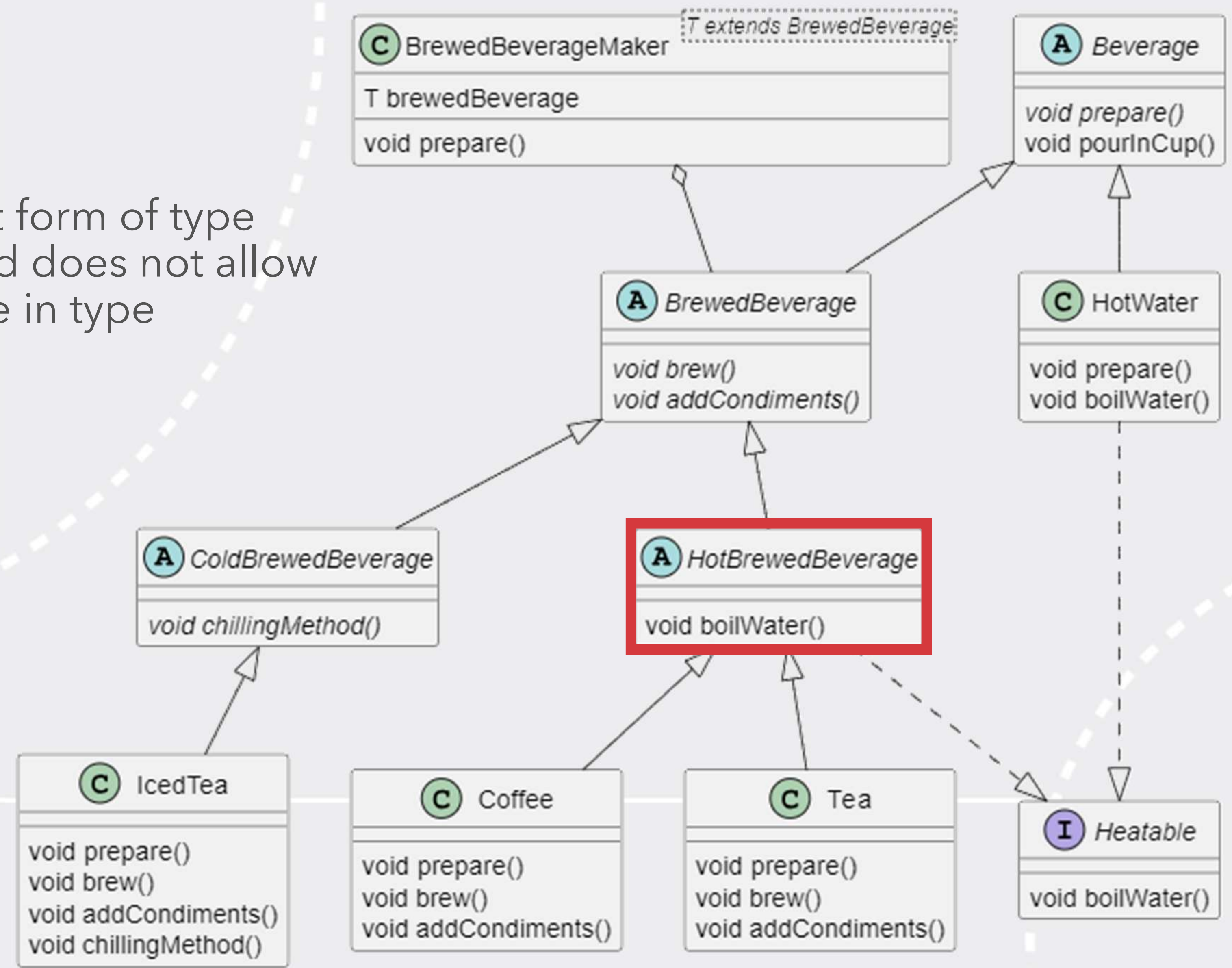
```
void prepareHotBrewedBeverage(Collection<? super HotBrewedBeverage> hotBrewedBeverages) {
…
}

Collection<? extends Beverage> anyBeverages = BeverageFactory.getAnyBeverageOrders();
Collection<? extends BrewedBeverage> anyBrewedBeverages = getAnyBrewedBeverageOrders();
Collection<BrewedBeverage> onlyBrewedBeverages = getBrewedBeverageOrders();
Collection<HotBrewedBeverage> hotBrewedBeverages = getHotBrewedBeverageOrders();


prepareHotBrewedBeverage(anyBeverages);
prepareHotBrewedBeverage(anyBrewedBeverages);
prepareHotBrewedBeverage(onlyBrewedBeverages);
prepareHotBrewedBeverage(hotBrewedBeverages);
```

INVARIANCE (THE RELATIONSHIP IS FIXED)

It is the strictest form of type relationship and does not allow for any variance in type relationships



Trial Lecture

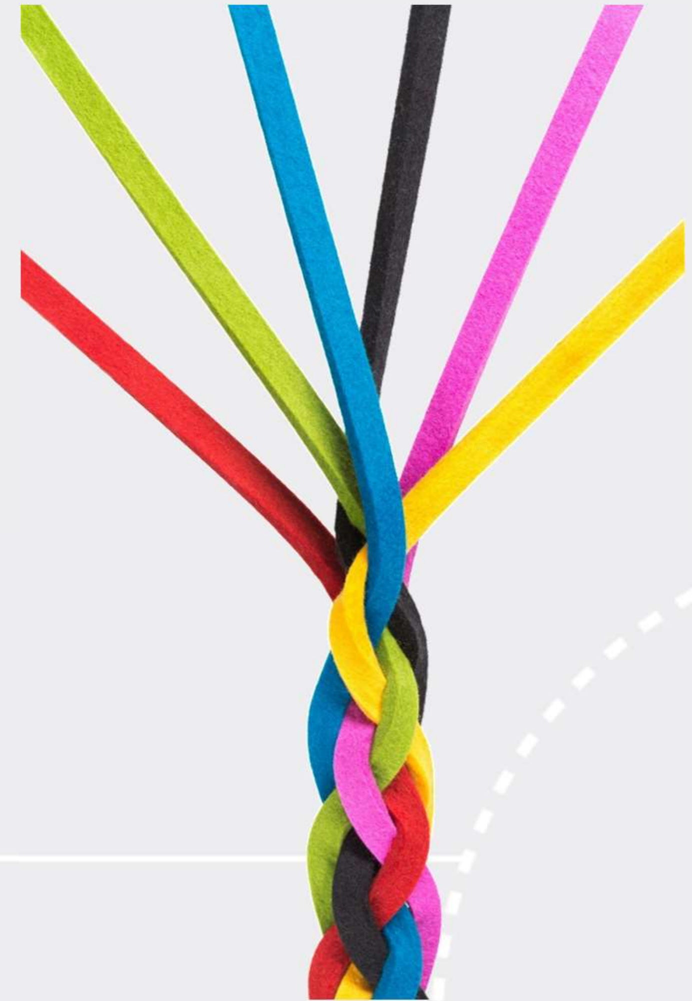
DESIGN PATTERNS[6]

- A general **reusable solution to a commonly occurring problem** in software design.
- **A template or blueprint** that can be applied to various situations to solve specific design problems in a consistent and efficient way
- Help developers create software that is more
  - **maintainable, scalable, and flexible**
- Some popular design patterns include Singleton, Factory, Observer, and Template patterns.

DESIGN PATTERS AND OOP

- These patterns not only tackle recurring issues but also **embody principles of good object-oriented programming**.

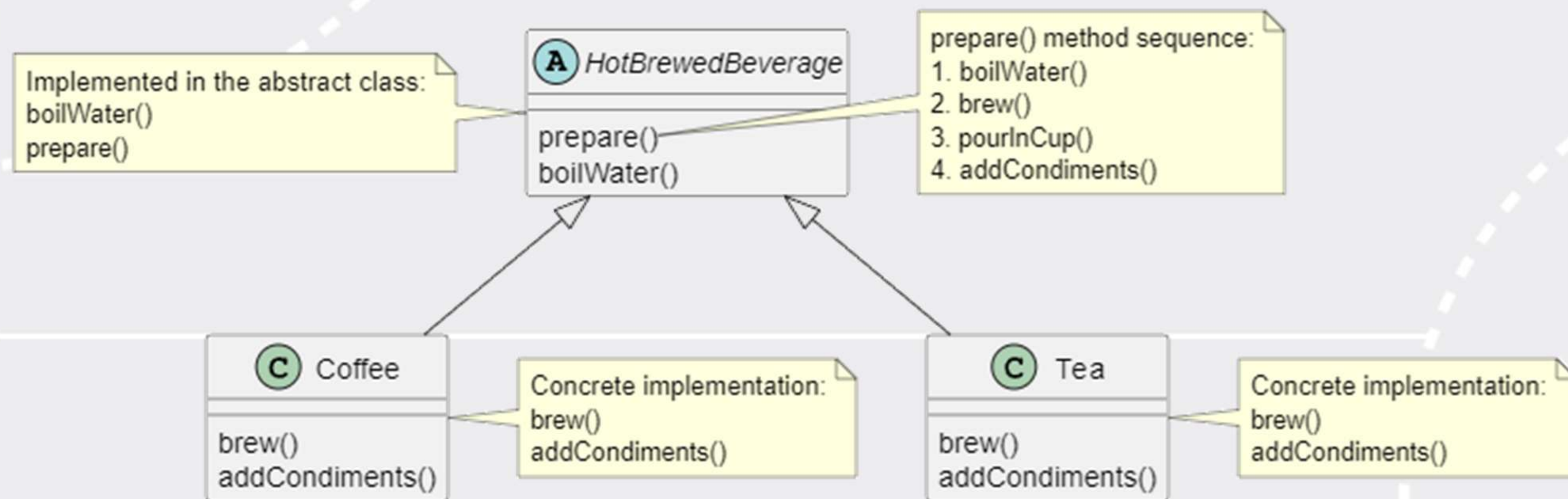
- Design patterns and object-oriented programming are **intertwined**, with design patterns serving as **practical applications of OOP concepts**.



Source: Microsoft PowerPoint

TEMPLATE DP

- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- It allows subclasses to modify certain steps of the algorithm without changing its structure.



TEMPLATE DP: ABSTRACTION, INHERITANCE

Algorithm's skeleton

```
abstract class HotBrewedBeverage … {
    @Override
    public void prepare() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    @Override
    public void boilWater() {
           …
    }
}
```

Override

```
class Coffee extends HotBrewedBeverage {
    void brew() {
        System.out.println("Dripping Coffee…");
    }

    void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}

class Tea extends HotBrewedBeverage {
    void brew() {
        System.out.println("Steeping the tea");
    }

    void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

TEMPLATE DP: GENERICS, POLYMORPHISM

```
class BeverageMaker<T extends Beverage> {
    T beverage;

    BeverageMaker(T beverage) {
        this.beverage = beverage;
    }

    void prepare() {
        beverage.prepare();
    }
}
```

```
BeverageMaker<Tea> teaMaker = new BeverageMaker<>(new Tea());
teaMaker.prepare();

BeverageMaker<Coffee> coffeeMaker = new BeverageMaker<>(new Coffee());
coffeeMaker.prepare();
```

CONCLUDING REMARKS

- Inheritance vs Composition
  - Composition is when you add functionality by referencing other objects
  - Relationship
    - Composition represents a "has-a" relationship
    - Inheritance represents an "is-a" relationship
- Inheritance & composition are two main techniques for code reuse in OOP.
- "Favor composition over inheritance"[6]
  - favouring composition over inheritance makes objects and classes more reusable, independent, loosely coupled and focused on single responsibilities.
  - Inheritance should only be used when composition cannot achieve the required behavior.

SUMMARY

- **Inheritance**
  - Allows classes to inherit properties and behaviors from base classes in a hierarchical relationship.
- **Polymorphism**
  - Allows objects of different classes to be treated as objects of a common superclass.
  - Key advanced concepts like **covariance, contravariance** were explained in the context of inheritance and polymorphism.
- **Design patterns**
  - Provide general reusable solutions to commonly occurring problems in software design.
  - The template pattern was demonstrated as an example.
- The significance of these advanced OOP concepts is that they help **create reusable, adaptable and maintainable** software, which are important for robust large-scale applications.
- The objective of the lecture was to gain a **deeper understanding** of these concepts and learn how to properly **apply** them when designing object-oriented programs.

REFERENCES

- [1] Oracle (2022), Oracle Java Documentation. Retrieved Mars 14, 2024, from https://docs.oracle.com/javase/tutorial/java/concepts/object.html
- [2] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. In ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7 (pp. 406-431). Springer Berlin Heidelberg.
- [3] Barnes, D. J., Kölling, M., & Gosling, J. (2006). Objects First with Java: A practical introduction using BlueJ (p. 520). Pearson/Prentice Hall.
- [4] Powered by AI and the LinkedIn community (2024). What is type theory and how is it used in programming languages? Retrieved Mars 10, 2024, from https://www.linkedin.com/advice/3/what-type-theory-how-used-programming-languages-mjy0f?trk=public_post_main-feed-card_feed-article-content
- [5] Veerpal Brar (June 30, 2021). Inheritance Vs Composition. Retrieved Mars 12, 2024, from https://veerpalbrar.github.io/blog/2021/06/30/Inheritance-vs-Composition
- [6] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH.
- [7] Eliza Taylor (February 8, 2024). 4 Principles of Object-Oriented Programming. Retrieved Mars 14, 2024, from https://www.theknowledgeacademy.com/blog/principles-of-object-oriented-programming/
- [8] ABHAY S. (April 27, 2023). The Importance of Object-Oriented Programming (OOP) in Java. Retrieved Mars 14, 2024 from https://www.linkedin.com/pulse/importance-object-oriented-programming-oop-java-abhay-singh/

Designed by: Freepik

INVARIANCE: EXAMPLE

```
BrewedBeverageMaker<HotBrewedBeverage> hotBrewedBeverageMaker =
          new BrewedBeverageMaker<>(new Coffee());


BrewedBeverageMaker<Coffee> coffeeMaker2 =
          (BrewedBeverageMaker<Coffee>)hotBrewedBeverageMaker; // Compile time error
```