Java Project – Hogwarts Legacy

- The package name for your project has to be **aMatriculationNumber** (e.g. a12345678).
- Instance variables must be declared `private`.
- The signatures of the specified functions must not be changed.
- The Java version on almighty is Java 17; use this version to avoid problems during the test!

For the test you have to upload the **Basic Implementation**. Be sure to store your basic implementation before you start working on the extension for the bonus point. **Upload of basic implementation until 24.01.2024**

1 Basic Implementation

1.1 Terminology

In the following we use the term `mana` for magic energy. Magic energy is needed to cause magical effects (e.g. casting a spell).

We use the term `health` to describe the health of living objects or the durability of inanimate objects. If an object reaches health 0 it is killed or destroyed.

The amount of mana that is available to an object is abbreviated to MP (mana points), the amount of an object's remaining health is abbreviated to HP (health points or hit points). Both, MP and HP can never be negative.

For the basic implementation you have to implement the following classes and interfaces:

1.2 Enum MagicLevel

This enum defines the constants `NOOB`, `ADEPT`, `STUDENT`, `EXPERT` and `MASTER`. Each constant is associated to a basic MP value (int) that constitutes the default for a wizard of that specific level. A method `toMana` has to be provided that returns the basic MP value associated to the enum constant. The `toString` method has to be overridden in a way that results in a string containing a number of asterisk characters respective to the level. The following table lists the MP values and string conversions for each enum constant:

constant	MP	toString
NOOB	50	"*"
ADEPT	100	"**"
STUDENT	200	"***"
EXPERT	500	"****"
MASTER	1000	"*****"

1.3 Interface MagicSource

```
public interface MagicSource {
  boolean provideMana(MagicLevel levelNeeded, int manaAmount);
    //levelNeeded==null or negative manaAmount must throw
        IllegalArgumentException; returns true if the object has at least
        the required level and can provide enough mana, false otherwise.
    //a typical implementation will check if the objects level is high
        enough, returning false if not. Otherwise it reduces the object's
        MP by manaAmount. There may be exceptions though, like objects with
         infinite mana or supporting all levels  for example
}
```

1.4 Interface MagicEffectRealization

Note that this interface provides default implementations for all methods. So a class implementing the interface only has to implement the realizations for effects that can affect the respective class instances.

```
public interface MagicEffectRealization {
  default void takeDamage(int amount) {}
    //negative amount must throw IllegalArgumentException;
    //a typical implementation will reduce the object's HP by amount
        ensuring however that HP does not become negative.

  default void takeDamagePercent(int percentage) {}
    //percentage must be between 0 and 100 (inclusive) otherwise an
        IllegalArgumentException must be thrown;
    //a typical implementation will reduce the object's HP by the
        percentage given of the object's basic HP value ensuring however,
        that HP does not become negative.
    //calculations must be done in double data type converting back to int
         only in the last step

  default void weakenMagic(int amount) {}
    //negative amount must throw IllegalArgumentException;
    //a typical implementation will reduce the object's MP by amount
        ensuring however that MP does not become negative.

  default void weakenMagicPercent(int percentage) {}
    //percentage must be between 0 and 100 (inclusive) otherwise an
        IllegalArgumentException must be thrown;
    //a typical implementation will reduce the object's MP by the
        percentage given of the object's basic MP value value ensuring
        however, that MP does not become negative..
    //calculations must be done in double data type converting back to int
         only in the last step
```

```
  default void heal(int amount) {}
    //negative amount must throw IllegalArgumentException;
    //a typical implementation will increase the object's HP by the amount
        given.

  default void healPercent(int percentage) {}
    //percentage must be between 0 and 100 (inclusive) otherwise an
       IllegalArgumentException must be thrown;
    //a typical implementation will increase the object's HP by the
       percentage given of the object's basic HP value.
    //calculations must be done in double data type converting back to int
        only in the last step

  default void enforceMagic(int amount) {}
    //negative amount must throw IllegalArgumentException;
    //a typical implementation will increase the object's MP by the amount
        given.

  default void enforceMagicPercent(int percentage) {}
    //percentage must be between 0 and 100 (inclusive) otherwise an
       IllegalArgumentException must be thrown;
    //a typical implementation will increase the object's MP by the
       percentage given of the object's basic MP value
    //calculations must be done in double data type converting back to int
        only in the last step

  default boolean isProtected(Spell s) {return false;}
    //if s is null, an IllegalArgumentException must be thrown;
    //an implementation returns true if the object is protected against
       the spell s, false otherwise.

  default void setProtection(Set<AttackingSpell> attacks) {}
    //if attacks is null an IllegalArgumentException must be thrown;
    //a typical implementation will register the object as protected
       against all the spells in attacks

  default void removeProtection(Set<AttackingSpell> attacks) {}
    //if attacks is null an IllegalArgumentException must be thrown;
    //a typical implementation will register the object as not protected
       against all the spells in attacks
}
```

1.5 Interface Trader

Objects of classes implementing this interface can trade (buy and sell) magic items. These objects manage some sort of inventory with a maximum capacity (weight) and are able to pay or receive payment in the relevant currency (i. e. Knuts).

```
public interface Trader {
  boolean possesses(Tradeable item);
    //if item is null an IllegalArgumentException must be thrown;
    //returns true if this object possesses the item, false otherwise

  boolean canAfford(int amount);
    //if amount is negative, an IllegalArgumentException must be thrown;
    //returns true if the object has enough money, false otherwise

  boolean hasCapacity(int weight);
    //if weight is negative, an IllegalArgumentException must be thrown;
    //returns true if the weight can be added to the object's inventory
        without exceeding the maximum weight capacity

  boolean pay(int amount);
    //if amount is negative, an IllegalArgumentException must be thrown;
    //if this owns enough money, deduct amount from money and return true,
         return false otherwise

  boolean earn(int amount);
    //if amount is negative, an IllegalArgumentException must be thrown;
    //add amount to this object's money and return true

  boolean addToInventory(Tradeable item);
    //if item is null, an IllegalArgumentException must be thrown;
    //if inventory capacity would be exceeded, return false
    //adds item to the inventory and returns true on success, false if
        adding the item failed

  boolean removeFromInventory(Tradeable item);
    //if item is null, an IllegalArgumentException must be thrown;
    //removes item from the inventory and returns true on success, false
        if removing the item failed

  default boolean canSteal() {
    //returns true if the object is capable to steal;
    //default implementation always returns false
    //this will be overridden for the class Wizard, so that only living
        Wizards can steal
  }

  boolean steal(Trader thief);
    //if thief is null, an IllegalArgumentException must be thrown;
    //if thief cannot steal (canSteal returns false) no action can be
        taken and false is returned
    //returns false if the object's inventory is empty
    //otherwise transfers a random item from the this object's inventory
        into the thief's inventory;
    //if the thief's inventory has not enough capacity, the object just
        vanishes and false is returned
    //returns true if theft was successful
```

```
  default boolean isLootable() {
    //returns true if the object can be looted;
    //default implementation always returns false
    //this will be overridden for the class Wizard, so that dead Wizards
        may be looted
  }

  default boolean canLoot() {
    //returns true if the object is capable to loot;
    //default implementation always returns false
    //this will be overridden for the class Wizard, so that only living
        Wizards can loot
  }

  boolean loot(Trader looter);
    //if looter is null, an IllegalArgumentException must be thrown;
    //if looter cannot loot (canLoot returns false) no action can be taken
         and false is returned
    //if the object can be looted (isLootable), transfer all the items in
        the object's inventory into the looter's inventory;
    //items that don't fit in the looter's inventory because of the weight
         limitation just vanish
    //returns true if at least one item was successfully transferred,
        false otherwise
}
```

1.6 Interface Tradeable

Objects that can be traded between Traders (see next section) typically have a price and a weight.

```
public interface Tradeable {
  int getPrice();
    //returns price of the object

  int getWeight();
    //returns weight of the object

  private boolean transfer(Trader from, Trader to) {
    //caution: this method transfers the item from from's inventory to to'
        s inventory without any checks. It has to be ensured that all
        necessary conditions for the transfer are met before calling this
        function.
    //The default implementation calls removeFromInventory on from and
        addToInventory on to and returns true if both calls succeeded (
        returned true)
  }

  default boolean give(Trader giver, Trader taker) {
    //if giver or taker is null or they are the same object, an
        IllegalArgumentException must be thrown;
    //giver gives the object away for free. Default implementation checks
        if the giver has the object (possesses method) and the taker has
        enough capacity in the inventory (hasCapacity). If any of these
        checks fail, the method returns false.
    //Otherwise the item is transferred from the giver's inventory to the
        taker's inventory (transfer method) and the return value of the
        transfer call is returned
  }

  default boolean purchase(Trader seller, Trader buyer) {
    //if seller or buyer is null or they are the same object, an
        IllegalArgumentException must be thrown;
    //default implementation checks if the seller has the object (
        possesses method), the buyer can afford the object (canAfford
        method) and the buyer has enough capacity in the inventory (
        hasCapacity). If any of these checks fail, the method returns false
        .
    //Otherwise the buyer pays the price (pay method), the seller receives
         the price paid (earn method), The item is transferred from the
        seller's inventory to the buyer's inventory (transfer method) and
        the return value of the transfer call is returned
  }

  void useOn(MagicEffectRealization target);
    //if target is null, an IllegalArgumentException must be thrown;
    //use the object on the target
}
```

1.7 Class MagicItem

This is an abstract base class for magic items. Magic items may be effected by magic and thus implement the `MagicEffectRealization` interface. The base class only defines an effect for the `takeDamagePercent` method. Classes deriving from `MagicItem` may implement additional methods or even supply their own overload for `takeDamagePercent`. As a general rule magic items will always provide enough mana to evoke the magic effects bound to them. This rule may be changed for specific subclasses.

```
public abstract class MagicItem implements Tradeable,
    MagicEffectRealization, MagicSource {
  private String name; //must not be null or empty
  private int usages; //number of usages remaining; must not be negative
  private int price;  //must not be negative
  private int weight; //must not be negative

  public int getUsages() {
    //returns value of usages (for access from deriving classes)
  }

  public boolean tryUsage() {
    //if usages > 0 reduce usage by 1 and return true, otherwise return
        false
  }

  public String usageString() {
    //returns "use" if usages is equal to 1, "uses" otherwise
  }

  public String additionalOutputString() {
    //returns empty string. Is overridden in deriving classes as needed
  }

  @Override
  public String toString() {
    //formats this object according to "['name'; 'weight' g; 'price' '
        currencyString'; 'usages' 'usageString''additionalOutputString']"
    //'currencyString' is "Knut" if price is 1, "Knuts" otherwise
    //e.g. (when additionalOutput() returns an empty string) "[Accio
        Scroll; 1 g; 1 Knut; 5 uses]" or "[Alohomora Scroll; 1 g; 10 Knuts;
         1 use]"
  }
```

```
  //Tradeable Interface:
  @Override
  public int getPrice() {
    //returns price of the object
  }

  @Override
  public int getWeight() {
    //returns weight of the object
  }

  //MagicSource Interface:
  @Override
  public boolean provideMana(MagicLevel levelNeeded, int amount) {
    //always returns true; no Exceptions needed
  }

  //MagicEffectRealization Interface:
  @Override
  public void takeDamagePercent(int percentage) {
    //reduce usages to usages*(1-percentage/100.)
  }
}
```

1.8 Class Potion

This class is just another abstract base for the various types of potions. It overrides the usageString method so that the result of the toString method will use "gulps" instead of "uses".

```
public abstract class Potion extends MagicItem {
  public void drink(Wizard drinker) {
    //delegates to method call useOn(drinker)
  }

  @Override
  public String usageString() {
    //returns "gulp" if usages is equal to 1, "gulps" otherwise
  }
}
```

1.9 Class HealthPotion

A potion that will increase the HP of the consumer

```
public class HealthPotion extends Potion {
  private int health;  //must not be negative

  @Override
  public String additionalOutputString() {
    //returns "; +'health' HP";
    //e.g. (total result of toString) "[Health Potion; 1 g; 1 Knut; 5
        gulps; +10 HP]"
  }

  @Override
  public void useOn(MagicEffectRealization target) {
    //if usages>0, reduce usages by 1 (tryUsage method) and
    //increase HP of target by health (call method heal(health))
  }
}
```

1.10 Class ManaPotion

A potion that will increase the MP of the consumer

```
public class ManaPotion extends Potion {
  private int mana; //must not be negative

  @Override
  public String additionalOutputString() {
    //returns "; +'mana' MP";
    //e.g. (total result of toString) "[Mana Potion; 1 g; 2 Knuts; 1 gulp;
         +20 MP]"
  }

  @Override
  public void useOn(MagicEffectRealization target) {
    //if usages>0 reduce usages by 1 (tryUsage method) and
    //increase MP of target by mana (call method enforceMagic(mana))
  }
}
```

1.11 Class Concoction

Concoctions can change HP and MP. They can also be reduced if the corresponding value is negative. Additionally an arbitrary number of spells is activated.

```
public class Concoction extends Potion {
  private int health;  //may be any int value
  private int mana;    //may be any int value
  private List<Spell> spells; //must not be null but may be empty; Use
      ArrayList as concrete type
  //it is not allowed for health and mana to be both 0 and spells to be
      empty; The potion must at least have one effect

  @Override
  public String additionalOutputString() {
    //returns "; '+/-''health' HP; '+/-''mana' MP; cast 'spells'";
    //here '+/-' denotes the appropriate sign, spells will be a bracketed
        list of spells (Java default toString method for lists)
    //e.g. (total result of toString) "[My Brew; 2 g; 2 Knuts; 4 gulps; -5
         HP; +10 MP; cast [[Confringo(*) -20 HP], [Diffindo(*) -15 HP]]]"
    //if health or mana is 0 or spells is empty, then the respective part(
        s) are suppressed e. g. "[Your Brew; 2 g; 1 Knut; 1 gulp; +5 MP]"
  }

  @Override
  public void useOn(MagicEffectRealization target) {
    //if usages>0 reduce usages by 1 (tryUsage method) and
    //change HP of target by health (call method heal(health) or
        takeDamage(health) depending on sign of health)
    //change MP of target by mana (call method enforceMagic(magic) or
        weakenMagic(magic) depending on sign of mana)
    //call cast Method for every spell in spells
  }
}
```

1.12 Class Spell

This class is an abstract base for the various types of spells.

```
public abstract class Spell {
  private String name;  //must not be null or empty
  private int manaCost; //must not be negative
  private MagicLevel levelNeeded; //must not be null

  public void cast(MagicSource source, MagicEffectRealization target) {
    //ensure necessary magic level and get necessary energy by calling
        provideMana on source (this will typically reduce MP in source)
    //if provideMana fails (returns false) cast is canceled
    //otherwise the abstract method doEffect is called
  }

  public abstract void doEffect(MagicEffectRealization target);
    //the actual effect of the spell on target must be implemented by the
        subclasses

  public String additionalOutputString() {
    //returns ""; is overridden in deriving classes when needed
  }

  @Override
  public String toString() {
    //return output in format "['name'('levelNeeded'): 'manaCost' mana'
        additionalOutputString']"; where 'levelNeeded' is displayed as
        asterisks (see MagicLevel.toString)
    //e.g. (full Output containing additionalOutputString) [Episkey(*): 5
        mana; +20 HP]
  }
}
```

1.13 Class AttackingSpell

Attacking spells cause damage by reducing HP or MP; if type is true HPs are reduced, otherwise MPs are reduced; The flag percentage is true if amount is a percentage value, false if amount is an absolute value

```
public class AttackingSpell extends Spell {
  private boolean type;
  private boolean percentage;
  private int amount; //has to be non negative; if percentage==true,
      amount must be in the interval [0,100]

  @Override
  public void doEffect(MagicEffectRealization target) {
    //if the target is protected against this spell (isProtected), then
        protection against exactly this spell is removed (removeProtection)
        .
    //otherwise use one of the functions takeDamage, takeDamagePercent,
        weakenMagic or weakenMagicPercent on target according to the flags
        type and percentage
  }

  @Override
  public String additionalOutputString() {
    //returns "; -'amount' 'percentage''HPorMP'", where 'percentage' is a
        percentage sign plus a space ("% ") if percentage is true, empty
        otherwise and HPorMP
    //is HP if type is true, MP otherwise
    //e. g. "; -10 MP" or "; -50 % HP"
  }
}
```

1.14 Class HealingSpell

Healing spells can restore HP or MP; if type is true HPs are restored, otherwise MPs are restored; The flag percentage is true if amount is a percentage value, false if amount is an absolute value

```
public class HealingSpell extends Spell {
  private boolean type;
  private boolean percentage;
  private int amount; //has to be non negative; if percentage==true,
      amount must be in the interval [0,100]

  @Override
  public void doEffect(MagicEffectRealization target) {
    //use one of the functions heal, healPercent, enforceMagic or
        enforceMagicPercent according to the flags type and percentage
  }

  @Override
  public String additionalOutputString() {
    //returns "; +'amount' 'percentage''HPorMP'", where 'percentage' is a
        percentage sign plus a space ("% ") if percentage is true, empty
        otherwise and HPorMP is HP
    //if type is true, MP otherwise
    //e. g. "; +10 HP" or "; +50 % MP"
  }
}
```

1.15 Class ProtectingSpell

Protecting spells create a shield against specific attacking spells using the setProtection method on Target

```
public class ProtectingSpell extends Spell {
  private Set<AttackingSpell> attacks; //must not be null or empty; use
      HashSet as concrete type

  @Override
  public void doEffect(MagicEffectRealization target) {
    //call setProtection method on target with attacks as parameter
  }

  @Override
  public String additionalOutputString() {
    //returns "; protects against 'listOfAttackSpells'" where '
        listOfAttackSpells' is a bracketed list of all the attack spells (
        Java default toString method for sets)
    //e. g. "; protects against [[Confringo(*): 10 mana; -20 HP], [
        Bombarda(**): 20 mana; -50 % HP]]"
  }
}
```

1.16 Class Scroll

Scrolls allow to cast spells of any magic level without having to use mana (the necessary energy is provided by the scroll)

```
public class Scroll extends MagicItem {
  private Spell spell;  //must not be null

  @Override
  public String additionalOutputString() {
    //returns "; casts 'spell'"
    //e.g. (total result of toString) "[Scroll of doom; 1 g; 100 Knuts; 5
        usages; casts [Bombarda(**): 20 mana; -50 % HP]]"
  }

  @Override
  public void useOn(MagicEffectRealization target) {
    //if usages>0 reduce usages by 1 (tryUsage method) and
    //cast the spell using this as magic source and parameter target as
        target
  }
}
```

1.17 Class Wizard

Wizards are the acting persons, they can trade items and use magic to fight each other

```
public class Wizard implements MagicSource, Trader, MagicEffectRealization
    {
  private String name; //not null not empty
  private MagicLevel level; //not null
  private int basicHP; //not negative
  private int HP; //not negative; defaults to basicHP
  private int basicMP; //not less than the manapoints associated with the
      magic level
  private int MP; //not negative; defaults to basicMP
  private int money; //not negative
  private Set<Spell> knownSpells; //not null, may be empty; use HashSet
      for instantiation
  private Set<AttackingSpell> protectedFrom; //not null, may be empty; use
       HashSet for instantiation
  private int carryingCapacity; //not negative
  private Set<Tradeable> inventory; //not null, may be empty, use HashSet
      for instantiation, total weight of inventory may never exceed
      carryingCapacity

  public boolean isDead() {
    //return true, if HP is 0, false otherwise
  }

  private int inventoryTotalWeight() {
    //calculates and returns the total weight of all the items in the
        inventory
  }

  public boolean learn(Spell s) {
    //if spell is null, IllegalArgumentException has to be thrown
    //if wizard is dead (isDead) no action can be taken and false is
        returned
    //add spell to the set of knownSpells
    //returns true, if insertion was successful, false otherwise
  }

  public boolean forget(Spell s) {
    //if spell is null, IllegalArgumentException has to be thrown
    //if wizard is dead (isDead) no action can be taken and false is
        returned
    //remove spell from the set of knownSpells
    //returns true, if removal was successful, false otherwise
  }
```

```
public boolean castSpell(Spell s, MagicEffectRealization target) {
  //if s or target is null, IllegalArgumentException has to be thrown
  //if wizard is dead (isDead) no action can be taken and false is
      returned
//if wizard does not know the spell, false is returned
  //call cast on s with this as source and parameter target as target
  //return true, if cast was called
}

public boolean castRandomSpell(MagicEffectRealization target) {
  //if this object's knownSpells is empty, return false
  //otherwise choose a random spell from knownSpells and delegate to
      castSpell(Spell, MagicEffectRealization)
}

public boolean useItem(Tradeable item, MagicEffectRealization target) {
  //if item or target is null, IllegalArgumentException has to be thrown
  //if wizard is dead (isDead) no action can be taken and false is
      returned
//if wizard does not possess the item, false is returned
  //call useOn on the item with parameter target as target
//return true if useOn was called
}

public boolean useRandomItem(MagicEffectRealization target) {
  //if this object's inventory is empty, return false
  //otherwise choose a random item from inventory and delegate to
      useItem(Tradeable, MagicEffectRealization)
}

public boolean sellItem(Tradeable item, Trader target) {
  //if item or target is null, IllegalArgumentException has to be thrown
  //if wizard is dead (isDead), no action can be taken and false is
      returned
  //call purchase on the item with this as seller and target as buyer
  //return true, if purchase was called successfully (returned true),
      false otherwise.
}

public boolean sellRandomItem(Trader target) {
  //if this object's inventory is empty, return false
  //otherwise choose a random item from inventory and delegate to
      sellItem(Tradeable, MagicEffectRealization)
}

@Override
public String toString() {
  //returns a string in the format "['name'('level'): 'HP'/'basicHP' 'MP
      '/'basicMP'; 'money' 'KnutOrKnuts'; knows 'knownSpells'; carries '
      inventory']"; where 'level' is the asterisks representation of the
      level (see MagicLevel.toString) and 'knownSpells' and 'inventory'
      use the default toString method of Java Set; 'KnutOrKnuts' is Knut
      if 'money' is 1, Knuts otherwise
  //e.g. [Ignatius(**): 70/100 100/150; 72 Knuts; knows [[Episkey(*): 5
      mana; +20 HP], [Confringo(*): 10 mana; -20 HP]]; carries []]
}
```

```java
//MagicSource Interface
@Override
public boolean provideMana(MagicLevel levelNeeded, int manaAmount) {
  //if wizard is dead (isDead) no action can be taken and false is
      returned
  //check if level is at least levelNeeded, return false otherwise
  //if MP<manaAmount return false
  //subtract manaAmount from MP and return true
}

//Trader Interface
@Override
public boolean possesses(Tradeable item) {
  //return true. if the item is in the inventory, false otherwise
}

@Override
public boolean canAfford(int amount) {
  //return true, if money>=amount, false otherwise
}

@Override
public boolean hasCapacity(int weight) {
  //return true, if inventoryTotalWeight+weight<=carryingCapacity, false
       otherwise
}

@Override
public boolean pay(int amount) {
  //if wizard is dead (isDead), no action can be taken and false is
      returned
  //if this owns enough money, deduct amount from money and return true,
       return false otherwise
}

@Override
public boolean earn(int amount) {
  //if wizard is dead (isDead), no action can be taken and false is
      returned
  //add amount to this object's money and return true
}

@Override
public boolean addToInventory(Tradeable item) {
  //add item to inventory if carryingCapacity is sufficient
  //returns true, if item is successfully added, false otherwise (
      carrying capacity exceeded or item is already in the inventory)
}

@Override
public boolean removeFromInventory(Tradeable item) {
  //remove item from inventory
  //returns true, if item is successfully removed, false otherwise (item
       not in the inventory)
}
```

```
@Override
public boolean canSteal() {
  //returns true, if this object's HP are not 0 (alive wizard)
}

@Override
public boolean steal(Trader thief) {
  //if thief is null, IllegalArgumentException has to be thrown
  //if thief cannot steal (canSteal returns false), no action can be
      taken and false is returned
  //returns false, if the object's inventory is empty
  //otherwise transfers a random item from the this object's inventory
      into the thief's inventory;
  //if the thief's inventory has not enough capacity, the object just
      vanishes and false is returned
  //returns true if, theft was successful
}

@Override
public boolean isLootable() {
  //returns true, if this object's HP are 0 (dead wizard)
}

@Override
public boolean canLoot() {
  //returns true, if this object's HP are not 0 (alive wizard)
}

@Override
public boolean loot(Trader looter) {
  //if looter is null, IllegalArgumentException has to be thrown
  //if looter cannot loot (canLoot returns false), no action can be
      taken and false is returned
  //if the this object can be looted (isLootable), transfer all the
      items in the object's inventory into the looter's inventory;
  //items that don't fit in the looter's inventory because auf the
      weight limitation just vanish
  //returns true, if at least one item was successfully transferred,
      false otherwise
}

//MagicEffectRealization Interface
@Override
public void takeDamage(int amount) {
  //reduce the object's HP by amount ensuring however that HP does not
      become negative.
}

@Override
public void takeDamagePercent(int percentage) {
  //reduce the object's HP by the percentage given of the object's basic
       HP value ensuring however, that HP does not become negative. Do
      calculations in double truncating to int only for the assignment
}
```

```
  @Override
  public void weakenMagic(int amount) {
    //reduce the object's MP by amount ensuring however that MP does not
        become negative.
  }

  @Override
  public void weakenMagicPercent(int percentage) {
    //reduce the object's MP by the percentage given of the object's basic
         MP value ensuring however, that MP does not become negative. Do
        calculations in double truncating to int only for the assignment
  }

  @Override
  public void heal(int amount) {
    //increase the object's HP by the amount given.
  }

  @Override
  public void healPercent(int percentage) {
    //increase the object's HP by the percentage given of the object's
        basic HP. Do calculations in double truncating to int only for the
        assignment
  }

  @Override
  public void enforceMagic(int amount) {
    //increase the object's MP by the amount given.
  }

  @Override
  public void enforceMagicPercent(int percentage) {
    //increase the object's MP by the percentage given of the object's
        basic MP. Do calculations in double truncating to int only for the
        assignment
  }

  @Override
  public boolean isProtected(Spell s) {
    //return true, if s is contained in instance variable protectedFrom
  }

  @Override
  public void setProtection(Set<AttackingSpell> attacks) {
    //add all spells from attacks to instance variable protectedFrom
  }

  @Override
  public void removeProtection(Set<AttackingSpell> attacks) {
    //remove all spells from attacks from instance variable protectedFrom
  }
}
```

1.18 Constructors

Your implementation has to provide at least one constructor for every class. This constructor has one parameter for each instance variable (in the order of their definition, starting with base classes), checks all integrity constraints and initializes all instance variables appropriately.

1.19 Tests

Create a Main-class with a main function in which you thoroughly test all the functionality of your implementation.

2 Additional Task

- If you would like to receive the additional point, please note that in this case an automatic **plagiarism check** is carried out for both the basic implementation and the additional implementation.
- Bonus points **do not** contribute to the test points required for passing the course, but may lead to an improvement in the grade.
- **Submission of the additional task until 30.06.2023**
- Remember to cache your base implementation (that you must upload for the third test) before you start adding.

To earn the additional bonus point, complete two out of the following three tasks:

2.1 Refactor Class Wizard

The constructor of the class `Wizard` has eleven parameters. A large number of constructor parameters is often considered a "code smell". The single responsibility principle is likely to be violated in such a class. This article describes various ways to reduce the number of constructor parameters for a class. Choose one of the methods described and use it to refactor your `Wizard` class. Give a short reason why you chose the specific method in a comment in your code.

2.2 Use the Visitor Pattern

In the specification the effect of a magic item depends on the classes of two objects: the magic item itself and the target that is affected. Since C++ only provides single dispatch (in contrary to double or multiple dispatch) there is a little trick used. All magic items provide a `useOn` method with the target as a parameter. Different possible targets all implement the `MagicEffectRealization` interface which in turn provides all the possibilities to magically influence the target. This construction uses both object's classes (magic item and target) to determine the desired effect. Since it uses inheritance however this is done at compile time and can prove inflexible sometimes. A more flexible way to achieve the same goal is provided by the visitor design pattern (see this Wikipedia article.

Refactor your implementation, so that the visitor pattern is used instead of inheritance.

2.3 Add State to `Wizard` Objects

Extend your implementation, so that wizard's can have one or more states like e.g. muted, blind, paralyzed. With a new type of spell it should be possible to affect the state of a wizard. If a wizard is inflicted with a state, he or she cannot take specific actions. For example a mute wizard cannot cast a spell but still use potions. A blind wizard may use spells but cannot read scrolls, etc. You may choose your own rules defining the exact limitations of a wizard in a specific state.