# Class 8: Interfaces, Generics, & Enumerations

**6.102 — Software Construction**
**Spring 2024**

## BasicBag

Nanoquiz: yellkey.com/**name**

Exercise: yellkey.com/**sport**

Pair up and get an exercise repo

Open `BasicBag.ts`

Look at the code

Write:

- **rep invariant** (hint: make consistent with `checkRep`)
- **abstraction function**
- **safety from rep exposure**

## Nanoquiz

- This quiz is just for you and your own brain:
  - closed-book, closed-notes
  - nothing else on your screen
- Lower your laptop screen when you’re done

yellkey.com/**name**

## BasicBag

Nanoquiz: yellkey.com/**name**

Exercise: yellkey.com/**sport**

Pair up and get an exercise repo

Open `BasicBag.ts`

Look at the code

Write:

- **rep invariant** (hint: make consistent with `checkRep`)
- **abstraction function**
- **safety from rep exposure**

```
class BasicBag {

    private elements: Array<string> = [];
```

**Which are good to include in the rep invariant?**
(pick all good choices, but only if they apply to this particular implementation)

```
    // Representation invariant:
    //   (A) `elements` is sorted in increasing order
    //   (B) `elements` cannot be empty
    //   (C) true
    //   (D) `elements` contains the elements of the bag
```

```
class BasicBag {
    private elements: Array<string> = [];
    // Representation invariant:
    //   true
```

**Which are good alternative ways to express the abstraction function?** (pick all that apply)

```
// Abstraction function:
//
// (A) AF(elements) = each member of the multiset is found in the array
//
// (B) AF(elements) = the multiset { elements[0], ..., elements[n-1] }
//                         where n = elements.length
//
// (C) AF(elements) = the multiset of all values found in elements
//
```

```
class BasicBag {
    private elements: Array<string> = [];
    // Representation invariant:
    //   true
    // Abstraction function:
    //   AF(elements) = the multiset of all values found in elements
```

**Which are good to include in the safety argument?** (pick all that apply; redundancy is okay)

```
    // Safety from rep exposure:
    //   (A) all fields are private
    //   (B) all fields are immutable
    //   (C) all public method arguments and return values are immutable
    //   (D) no public method takes or returns an array
    //   (E) checkRep() is called in every method

    public BasicBag() { ...checkRep()... }
    public size():number { ...checkRep()... }
    public contains(elt:string):boolean { ...checkRep()... }
    public add(elt:string):void { ...checkRep()... }
    public remove(elt:string):void { ...checkRep()... }
}
```

## Documented `BasicBag`

```
class BasicBag {

    private elements: Array<string> = [];

    // Representation invariant:
    //   true

    // Abstraction function:
    //   AF(elements) = the multiset of all values found in elements

    // Safety from rep exposure:
    //   all fields are private
    //   Array is the only mutable type used in the rep, and
    //       no public method takes or returns Array
}
```

## Make `BasicBag` a subtype of `Bag`

```
class BasicBag implements Bag
```

Extract the spec of `BasicBag` into interface `Bag`

- Put the **specs of operations** into `Bag`
- Leave the **rep** and **method implementations** behind in `BasicBag`

## Make `BasicBag` a subtype of `Bag`

```
class BasicBag implements Bag
```

Extract the spec of `BasicBag` into interface `Bag`

- Put the **specs of operations** into `Bag`
- Leave the **rep** and **method implementations** behind in `BasicBag`

Update the tests to use `Bag` instead of `BasicBag`

- **`BagTest` no longer mentions `BasicBag`** except for constructor `new BasicBag()`
- All tests should pass

## Make the bags generic

```
// A mutable bag of elements of type E.
// @template E the type of elements in the bag
interface Bag<E>
```

1. Make `Bag` generic, but keep `BasicBag` as a bag of strings for now

```
      class BasicBag implements Bag<____> { ... }
```

- Fix all issues and make the tests pass again

## Make the bags generic

```
// A mutable bag of elements of type E.
// @template E the type of elements in the bag
interface Bag<E>
```

1. Make `Bag` generic, but keep `BasicBag` as a bag of strings for now

```
      class BasicBag implements Bag<____> { ... }
```

- Fix all issues and make the tests pass again

2. Now make `BasicBag` generic, too

```
      class BasicBag<E> implements Bag<E> { ... }
```

- Fix all issues and make the tests pass again
- **`Bag` and `BasicBag` should no longer mention `string` at all**
- (Tests still only mention `BasicBag` to call the constructor)

## Provide a factory

```
// A mutable bag of elements of type E.
// @template E the type of elements in the bag
interface Bag<E>
```

1. Make `Bag` generic, but keep `BasicBag` as a bag of strings for now

2. Now make `BasicBag` generic, too

```
      class BasicBag<E> implements Bag<E> { ... }
```

3. At the bottom of `Bag.ts`, add a factory function:

```
export function makeBag<E>(): Bag<E> { return ... }
```

- What should it return?
- Use the factory function to remove all mentions of `BasicBag` from `BagTest.ts`

Example generic versions of `Bag`, `BasicBag`, and `BagTest`: yellkey.com/**local**

## Make a new subtype of `Bag`

```
// A mutable bag of coin flips.
class FlipBag implements Bag<Flip>
```

- uncomment the code in `FlipBag.ts`
- fix its TODOs
  - **including the AF of `FlipBag`**
- fix all issues and make all tests pass

## Which are good abstraction functions for `FlipBag`?

```
class FlipBag implements Bag<Flip> {
  private flips: number = 0;
  private heads: number = 0;
  // Representation invariant:
  //   heads and flips are both integers, 0 <= heads <= flips

  // Abstraction function AF(flips,heads) =
  //
  // (A) heads=heads and tails=flips-heads
  //
  // (B) the multiset consisting of `heads` occurrences of HEADS
  //                   and `flips`-`heads` occurrences of TAILS
  //
  // (C) { HEADS^heads, TAILS^(`flips`-`heads`) }
  //
  // (D) the number of heads is stored in `heads` and
  //     the number of total flips is stored in `flips`
...
}
```

Suppose we add a method to `Bag`:

```
interface Bag<E> {
    // @returns true iff the multiplicity of every element in `this`
    //          is less than or equal to its multiplicity in `that`
    public subBag(Bag<E> that): boolean;
    ...
}
class BasicBag<E> implements Bag<E> { ... }
class FlipBag implements Bag<Flip> { ... }
```

What does this do? (pick all that apply)

A. strengthens the spec of `Bag`
B. weakens the spec of `Bag`
C. requires changing `BasicBag`
D. requires changing `FlipBag`
E. requires reviewing/changing clients of `Bag`

Suppose we add a method to `FlipBag`:

```
interface Bag<E> { ... }
class BasicBag<E> implements Bag<E> { ... }
class FlipBag implements Bag<Flip> {
    // @param p probability of getting heads from some coin
    // @returns probability of flipping this bag's combination
    //          of heads and tails using that coin
    public probability(p: number): number;
    ...
}
```

What does this do? (pick all that apply)

A. strengthens the spec of `FlipBag`
B. weakens the spec of `FlipBag`
C. requires changing `Bag`
D. requires changing `BasicBag`
E. requires reviewing/changing clients of `Bag`

Suppose we change the spec of a method in `Bag`:

```
interface Bag {
    // Modifies this bag by removing one occurrence of elt, if found.
    // ~~If elt is not found in the bag, has no effect.~~
    // @param elt element to remove. Requires elt to be in the bag.
    public remove(elt: string): void;
    ...
}
class BasicBag implements Bag { ... }
class FlipBag implements Bag { ... }
```

What does this do? (pick all that apply)

A. strengthens the spec of `Bag`
B. weakens the spec of `Bag`
C. requires changing `BasicBag`
D. requires changing `FlipBag`
E. requires reviewing/changing clients of `Bag`

## The word “interface”

## The word “abstract”