Immutability and Selecting ADT's

CS61B Sp18 Discussion 6

Administrivia

- Midterm scores are out!
- Regrade requests are Friday the 23rd at noon. No regrades will be accepted later than that.
  - Please format your regrade according to the format specified on Piazza @1520
- Project 2 final version is out
  - Phase 1 due February 26
  - Phase 2 due March 5th
- It is highly recommended that you find a partner for this project. Partner forms are linked in the spec. Please email jzeitsoff@berkeley.edu if you fill out the form late!
- Homework 1 is out and due 2/21

Review: Generics

Generics is a way to allow your data structures to work with any type.

```
public class List<T>{...}
```

This is a generic class

```
List<String> = new ArrayList<String>();
```

Instantiation

```
public static <K, V> V get(Map61B<K, V> map, K key){}
```

Generic method

```
public static <K extends Comparable<K>, V> V get(Map61B<K,
V> map, K key){}
```

Type upper bound

Review: Autoboxing

Every primitive in Java has an Object counterpart. Many data structures cannot hold primitives, so we wrap the primitive in an object.

Primitive	Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Doube
boolean	Boolean
char	Character

Java is smart and often converts primitive types to their Object counterparts or Objects to their primitive counterparts automatically. This is called **autoboxing.**

Review: Exceptions

```
try{
    risky code
} catch (Exception e) {
    fix it hurry
} finally {
    shut it down
}
```

Run some code that may throw an exception

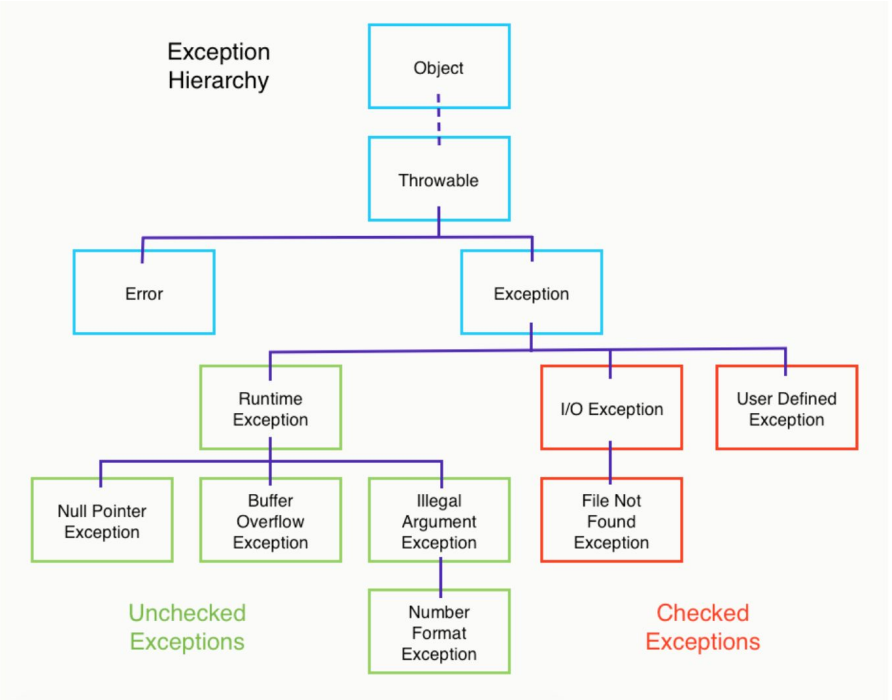
Will catch the exception if it is thrown in the try block

Will run no matter what.

Review: Types of Exceptions

There are 2 overarching types of exceptions:

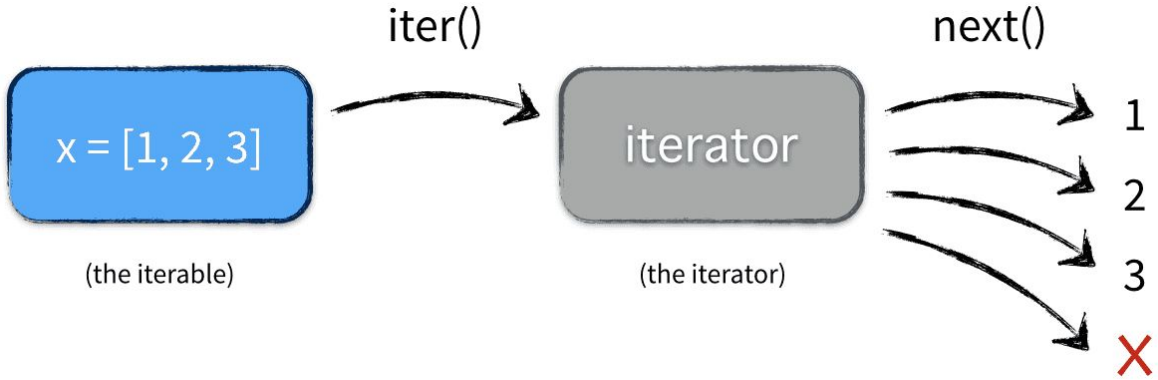
1. Checked
    a. You must either wrap them in a try/catch block or pass the buck by using “throws exception” in the method header
2. Unchecked
    a. You don’t need to handle these. They are typically an error on a user’s part, which can’t really be helped.



Review: Iterators and Iterables

- **Iterable:** Something that can be iterated over
  - Must have .iterator() method which creates an iterator
- **Iterator:** actually tool/machine you use to do the iteration.
  - Must have .next() and .hasNext() methods

Each iterable object will need to produce its own one-use iterator object.



Access Control

Access control allows us to restrict the use of methods, classes, and fields.

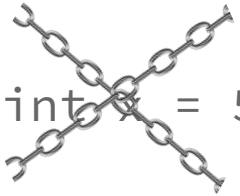
1. private 
2. protected 
3. default (no modifier) ¯_(ツ)_/¯
4. public 

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
	Y	Y	N	N
private	Y	N	N	N

Immutability

Immutable: A class or object is immutable if you cannot change anything about it after you instantiate it.

```
public static final int x = 5;
```



Now, nobody can change the value of x.

Caveat: If you instantiate a reference as final, then you are making the reference immutable, but not what is inside the object.

```
public static final List<String> arr = new ArrayList<String>()
```

- Although arr is declared as final, you can still call arr.insert("convfefe").
- However, you cannot reassign `arr = new LinkedList<String>();`

Abstract Data Types (Review from disc5)

Abstract data types are defined by their **behavior**, not by their **implementations.** AKA, they're about **what,** not **how.**

Some examples are:

- List: an ordered collection of elements
- Set: an unordered collection of unique elements (no repeats)
- Map: a collection of key/value pairs
- Stacks: has last in first out property (LIFO)
- Queues: has first in first out property (FIFO)
- Deques: can be both LIFO and FIFO

1.) Rocks pt1

```java
public class Pebble {
    public int weight;
    public Pebble() { weight = 1; }
}
```

```java
public class Rock {
    public final int weight;
    public Rock (int w) { weight = w; }
}
```

1.) Rocks pt1

```
1  public class Pebble {
2      public int weight;
3      public Pebble() { weight = 1; }
4  }
```

Everything in the pebble class is public and non-final. It is mutable.

```
1  public class Rock {
2      public final int weight;
3      public Rock (int w) { weight = w; }
4  }
```

The Rock class has one int weight and it is declared as final, so it is immutable

1.) Rocks pt2

```
1  public class Rocks {
2      public final Rock[] rocks;
3      public Rocks (Rock[] rox) { rocks = rox; }
4  }
```

```
1  public class SecretRocks {
2      private Rock[] rocks;
3      public SecretRocks(Rock[] rox) { rocks = rox; }
4  }
```

1.) Rocks pt2

```
1  public class Rocks {
2      public final Rock[] rocks;
3      public Rocks (Rock[] rox) { rocks = rox; }
4  }
```

The rock variable is static, but you can still modify the contents of rocks, thus it is mutable.

```
1  public class SecretRocks {
2      private Rock[] rocks;
3      public SecretRocks(Rock[] rox) { rocks = rox; }
4  }
```

There may be a reference to rocks somewhere outside of the class, so it is mutable.

1.) Rocks pt3

```
public class PunkRock {
    private final Rock[] band;
    public PunkRock (Rock yRoad) { band = {yRoad}; }
    public Rock[] myBand() {
        return band;
    }
}
```

```
public class MommaRock {
    public static final Pebble baby = new Pebble();
}
```

1.) Rocks pt3

```
1  public class PunkRock {
2      private final Rock[] band;
3      public PunkRock (Rock yRoad) { band = {yRoad}; }
4      public Rock[] myBand() {
5          return band;
6      }
7  }
```

It is possible to access and modify the contents of PunkRock's private array through its public myBand() method, so this class is mutable.

```
1  public class MommaRock {
2      public static final Pebble baby = new Pebble();
3  }
```

This class is mutable since Pebble has public variables that can be changed. For instance, given a MommaRock mr, you could mutate mr with mr.baby.weight = 5.

2.) Breaking the System pt1

Exploit the flaw by filling in the main method below so that it prints “Success” by causing BadIntegerStack to produce a NullPointerException.

```
1   public static void main(String[] args) {
```

2.) Breaking the System pt1

Exploit the flaw by filling in the main method below so that it prints “Success” by causing BadIntegerStack to produce a NullPointerException.

```
public static void main(String[] args) {
    try {
        BadIntegerStack stack = new BadIntegerStack();
        stack.pop();
    } catch (NullPointerException e) {
        System.out.println("Success!");
    }
}
```

2.) Breaking the System pt1

Exploit the flaw by filling in the main method below so that it prints “Success” by causing BadIntegerStack to produce a NullPointerException.

```
1  public static void main(String[] args) {

1      try {
2          BadIntegerStack stack = new BadIntegerStack();
3          stack.pop();
4      } catch (NullPointerException e) {
5          System.out.println("Success!");
6      }
7  }
```

The issue is, the bad stack is initialized as empty with top being null. So, if you pop from a newly initialized stack, you will have no “top” variable with which to extract top.value.

2.) Breaking the System pt2

Exploit another flaw in the stack by completing the main method below so that the stack appears infinitely tall.

```
1   public static void main(String[] args) {
```

2.) Breaking the System pt2

Exploit another flaw in the stack by completing the main method below so that the stack appears infinitely tall.

```
1   public static void main(String[] args) {
1       BadIntegerStack stack = new BadIntegerStack();
2       stack.push(1);
3       stack.top.prev = stack.top;
4       while (!stack.isEmpty()) {
5           stack.pop();
6       }
7       System.out.println("This print statement is unreachable!");
8   }
```

2.) Breaking the System pt2

Exploit another flaw in the stack by completing the main method below so that the stack appears infinitely tall.

```
1   public static void main(String[] args) {
1       BadIntegerStack stack = new BadIntegerStack();
2       stack.push(1);
3       stack.top.prev = stack.top;
4       while (!stack.isEmpty()) {
5           stack.pop();
6       }
7       System.out.println("This print statement is unreachable!");
8   }
```

If you set top.prev to itself, then every time you pop you will set top back to itself! This means you can keep popping but top will remain in the list.

2.) Breaking the System pt3

How can we change the BadIntegerStack class so that it won't throw NullPointerExceptions or allow ne'er-do-wells to produce endless stacks?

2.) Breaking the System pt3

How can we change the BadIntegerStack class so that it won't throw NullPointerExceptions or allow ne'er-do-wells to produce endless stacks?

Check that top is not null before popping and make top private.

3.) Design a parking lot

Design a ParkingLot package that allocates specific parking spaces to cars in a smart way. Decide what classes you'll need, and design the API for each. Time permitting, select data structures to implement the API for each class. Try to deal with annoying cases (like disobedient humans).

- Parking spaces can be either regular, compact, or handicapped-only.
- When a new car arrives, the system should assign a specific space based on the type of car.
- All cars are allowed to park in regular spots. Thus, compact cars can park in both compact spaces and regular spaces.
- When a car leaves, the system should record that the space is free.
- Your package should be designed in a manner that allows different parking lots to have different numbers of spaces for each of the 3 types.
- Parking spots should have a sense of closeness to the entrance. When parking a car, place it as close to the entrance as possible. Assume these distances are distinct.

Car

`public class Car`

- `public Car(boolean isCompact, boolean isHandicapped)`: creates a car with given size and permissions.
- `public boolean isCompact()`: returns whether or not a car can fit in a compact space.
- `public boolean isHandicapped()`: returns whether or not a car may park in a handicapped space.
- `public boolean findSpotAndPark(ParkingLot lot)`: attempts to park this car in a spot, returning true if successful.
- `public void leaveSpot()`: vacates this car's spot.

Spot

`private class Spot`

- The `Spot` class can be declared private and encapsulated by the `ParkingLot` class. Though it is private, and therefore not a part of our parking lot API, its methods are described here to give you an idea of how a `Spot` class might be implemented.
- `private Spot(String type, int proximity)`: creates a spot of a given type and proximity.
- `private boolean isHandicapped()`: returns true if this spot is reserved for handicapped drivers.
- `private boolean isCompact()`: returns true if this parking space can only accomodate compact cars.

Parking Lot

`public class ParkingLot`

- `public ParkingLot(int[] handicappedDistances, int[] compactDistances, int[] regularDistances)`: creates a parking lot containing `handicappedDistances.length` handicapped spaces, each with a distance corresponding to an element of `handicappedDistances`. Also initializes the appropriate compact and regular spaces.
- `public boolean findSpotAndPark(Car toPark)`: attempts to find a spot and park the given car. Returns false if no spots are available.
- `public void removeCarFromSpot(Car toRemove)`: records when a spot has been vacated, and makes the spot available for parking again.