Immutable Classes

Immutable Classes

Immutable class is when instantiated cannot be modified.
Wrapper classes and String class are immutable hence thread-safe.

Creating an Immutable Object

To create an immutable object you need to follow some simple rules:

- Don't add any setter method
- Declare all fields final and private
- If a field is a mutable object create defensive copies of it for getter methods
- If a mutable object passed to the constructor must be assigned to a field create a defensive copy of it
- Don't allow subclasses to override methods.

Defensive copy in getter methods

Without defensive copy mechanism

```
public class DateContainer {
  private final Date date;
  public DateContainer() {
      this.date = new Date();
  }
  public Date getDate() {
    return date;
  }
}
```

Test

```
  DateContainer dateContainer = new DateContainer();
  System.out.println(dateContainer.getDate());
  dateContainer.getDate().setTime(dateContainer.getDate().getTime() + 1000);
  System.out.println(dateContainer.getDate());
```

Note: DateContainer date is 1 second after

With defensive copy mechanism

```
public class DateContainer {
  private final Date date;
  public DateContainer() {
      this.date = new Date();
  }
  public Date getDate() {
    return new Date(date.getTime());
  }
}
```

Test

```
  DateContainer dateContainer = new DateContainer();
  System.out.println(dateContainer.getDate());
  dateContainer.getDate().setTime(dateContainer.getDate().getTime() + 1000);
  System.out.println(dateContainer.getDate());
```

Note: DateContainer date is not changed because we changed the copy not the original object

Defensive copy in constructor

If a mutable object passed to the constructor must be assigned to a field create a defensive copy of it

Without defensive copy mechanism

```
public class DateContainer {
  private final Date date;

  public DateContainer(Date date) {
      this.date = date;
  }

  public Date getDate() {
    return new Date(date.getTime());
  }
}
....
  Date date = new Date();
  DateContainer dateContainer = new DateContainer(date);
  System.out.println(dateContainer.getDate());
  date.setTime(date.getTime() + 1000);
  System.out.println(dateContainer.getDate());
```

Note: *Now dateContainer date is 1 second after also if the getter method*

With defensive copy/deep cloning mechanism

```
public class DateContainer {
  private final Date date;

  public DateContainer(Date date) {
      this.date = new Date(date.getTime());
  }

  public Date getDate() {
    return new Date(date.getTime());
  }
}
....
  Date date = new Date();
  DateContainer dateContainer = new DateContainer(date);
  System.out.println(dateContainer.getDate());
  date.setTime(date.getTime() + 1000);
  System.out.println(dateContainer.getDate());
```

Note: `Now dateContainer date is not changed. We create a copy on the constructor`

Pitfalls

1. Primitives :Primitives are immutable, so you don't have to do anything special.
2. Collections
3. Arrays :Java doesn't have any convenient methods to prevent arrays from being modified. Your best bet is to either hide the original array and always return a clone, or to not use arrays in the underlying implementation and instead convert a Collection object to an array.

Lists, arrays, maps, sets and other non-immutable objects can be surprising.

A private final object with no setter is fixed to the object it was initially assigned, but the values inside that object aren't fixed (unless the object is immutable).

That means you might have an

```
ImmutableShoppingList myShoppingList =
                                        new ImmutableShoppingList(new String[] {"apples", "oranges","pasta"})
```

and expect that the shopping list will always have "apples", "oranges" and "pasta".

Someone could call myShoppingList.getList()[0] = "candy bars"; and change your list to be "candy bars", "oranges" and "pasta", which is unhealthy and clearly not what you want.

Pitfalls : Solutions

java.util.Collections provides a number of convenience methods that make converting a Collection to an UnmodifiableCollection.

```
Collections.unmodifiableCollection()
Collections.unmodifiableList()
Collections.unmodifiableSet()
Collections.unmodifiableMap()
Collections.unmodifiableSortedMap()
Collections.unmodifiableSortedSet()
```

Without defensive copy mechanism/shallow cloning

```
public class ImmutableShoppingList {
    private final List<String> list;
    public ImmutableShoppingList(List<String> list){
        this.list = Collections.unmodifiableList(list);
    }
    public List<String> getList(){
        return list;
    }
}
```

With defensive copy mechanism/deep cloning

```
public class ImmutableShoppingList {
    private final List<String> list;

    public ImmutableShoppingList(List<String> list){
        List<String> tmpListOfHolding = new ArrayList<>();
        tmpListOfHolding.addAll(list);
        this.list = Collections.unmodifiableList(tmpListOfHolding);
    }
    public String[] getList(){
        return (String[]) list.toArray();
    }
}
```

Pitfalls : Solutions

Arrays :Java doesn't have any convenient methods to prevent arrays from being modified. Your best bet is to either hide the original array and always return a clone, or to not use arrays in the underlying implementation and instead convert a Collection object to an array.

```
public class ImmutableShoppingList {

    private final List<String> list;

    public ImmutableShoppingList(String[] list){
        this.list = Collections.unmodifiableList(Arrays.asList(list));
    }

    public String[] getList(){
        return (String[]) list.toArray();
    }
}
```

Modifying Immutable Object

Apply Builder design pattern:
The builder pattern creates a temporary object with the same fields as the desired object. It has getters and setters for all the fields. It also has a build() method that creates the desired object

```
//Immutable class
class ImmutableDog {
    private final String name;
    private final int weight

    public ImmutableDog(String name, int weight){
        this.name = name;
        this.weight = weight;
    }
    public String getName(){
        return this.name;
    }
    public int getWeight(){
        return this.weight;
    }
}
```

```
//Builder class
class ImmutableDogBuilder {
    private String name;
    private int weight;
    public ImmutableDogBuilder(){}
    public ImmutableDog build(){
        return new ImmutableDog(name, weight);
    }
    public ImmutableDogBuilder setName(String name){
        this.name = name;
        return this;
    }
    public ImmutableDogBuilder setWeight(int weight){
        this.weight = weight;
        return this;
    }
    public String getName(){
        return this.name;
    }
    public int getWeight(){
        return this.weight;
    }
}
```

```
ImmutableDogBuilder dogBuilder = new ImmutableDogBuilder().setName("Rover").setWeight(25);
```

Thank You!