[Handout for L8P1]
Putting up defenses to protect our code

Error handling

Exceptions and assertions are two complementary ways of handling errors in software.

Assertions are used to test your assumptions about the program. Using an assertion, you can say something like 'when the execution comes to this point, the variable n cannot be null'. That implies that the only way n can be null at that point is if there is a bug somewhere else in the program. If the runtime detect such an 'assertion failure', the system will halt with an error message. The reason is some buggy code has been executed and sooner the program stops, the safer it is.

```
int timeout = Config.getTimeout();  //line 1
assert timeout > 0 ;                //line 2
setTimeout(timeout);                //line 3
```

In the Java code given above, the assertion in line 2 verifies the assumption that timeout returned by Config.getTimeout() is greater than 0. The only reason for it to return a value of 0 or lower in line 1 is a bug in the system somewhere.

It is recommended that you should use assertions liberally in your code. If required, assertions can be disabled without modifying the code. For example, 'java -enableassertions HelloWorld' will run HelloWorld with assertions enabled while 'java -disableassertions HelloWorld' will run it without verifying assertions.

Notes:

1. The assertions mentioned above have a different purpose from the Assertions used in UnitTesting frameworks such as JUnit. The former are located *inside* the functional program and serve as verifiers of assumptions. The latter are located outside the functional program and used for explicit testing of the program.
2. Some runtime environments disable assertions by default. This could create a situation where the developer thinks all assertions are being verified as true while in fact they are not being verified at all. Therefore, remember to enable assertions when you run the program.

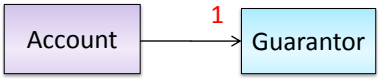
Exceptions on the other hand are used to deal with 'unusual' but not unexpected situations the program might encounter at runtime. For example, a network connection might timeout due to a slow server. That is not a program bug but rather an unusual situation we should try to recover from if possible. However, the method that encountered the unusual situation might not know how to recover from it. That is why most languages allow a method to encapsulate the unusual situation in an Exception object and 'throw' that object to the caller of the method to deal with it. If that caller method does not know how to deal with the exception it caught, the method will throw the Exception object to its own caller. If none of the callers is prepared to deal with the exception, the exceptions can propagate through the method call stack until it is received by the main method and thrown to the user, halting the system. One advantage of this throw-catch ability of exception is that you can separate code that deal with 'unusual' situations from the code that does the 'usual' work.

Therefore, Exceptions and assertions serve different purposes and you should use both in your code. Please find out by yourself how your preferred language supports exceptions and assertions.

Logging can be useful for troubleshooting problems that exceptions and assertions could not prevent (due to situations you failed to cater for). A good logging system records some system information regularly. When bad things happen to your system, you can use those log files to learn more about what went wrong and take actions to prevent it happening again. This is the same reason why airplanes have 'black boxes'. Find out how your preferred language environment supports logging. While you can implement your own logging system (e.g. by inserting file I/O statements to write to a log file), most programming environments come with logging systems that allow more sophisticated logging. For example, they allow you to turn logging on and off easily or to change to logging intensity (i.e. how much information to record).

Defensive programming

In defensive programming we assume that if we leave room for things to go wrong, they *will* go wrong. Therefore, a defensive programmer proactively tries to prevent things going wrong. Let us illustrate this using an example from. Consider two entities Account and Guarantor with an association as shown in the following diagram:

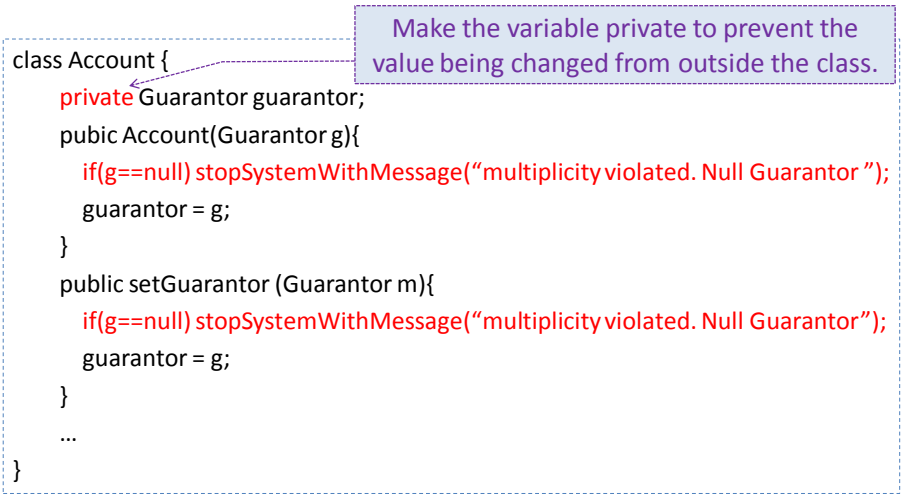



```
class Account {
    Guarantor guarantor;
    void setGuarantor(Guarantor g){
        guarantor = g;
    }
}
```

Here, the association is compulsory i.e. an Account object should always be linked to a Guarantor. One way we can implement this is to simply use a reference variable as above. However, what if someone else in the team used Account class as:

```
Account  a = new Account();
a. setGuarantor(null);
```

This results in an Account without a Guarantor! In a real banking system, this could have serious consequences!! The code here did not try to prevent such a thing from happening. To proactively enforce the multiplicity constraint, following offers a solution:



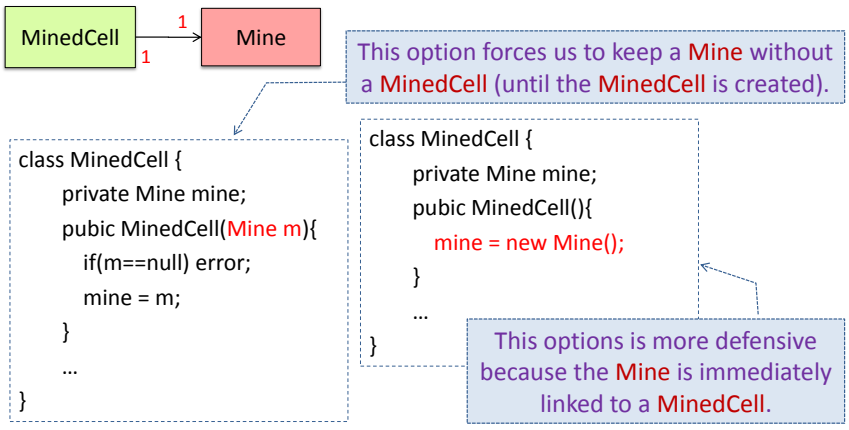
We need not be 100% defensive all the time. How defensive one should code depends on many factors such as:

- How critical is the reliability of the system?
- Will the code be used by programmers other than the author?
- The level of programming language support for defensive programming

Next, let us look at some more examples of defensive programming.

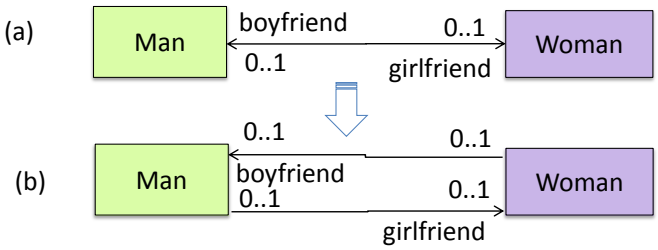
Enforcing 1-to-1 associations

Consider the association given below. Here, a MinedCell cannot exist without a Mine and vice versa. The only way to enforce this is by simultaneous object creation. But in Java and C++, you can create only one object at a time. Given below are two alternatives. Both options violate the multiplicity for a short period of time.



Enforcing referential integrity

A bidirectional association in the design is emulated at code level using two variables.



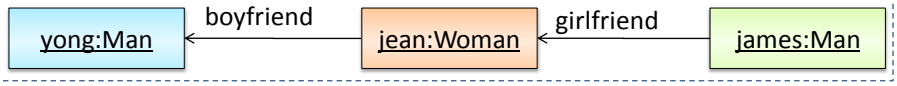

```
class Man {
      Woman girlfriend;
      void setGF(Woman w){
      girlfriend = w;
      }
      …
}
```

```
class Woman {
      Man boyfriend;
      void setBF(Man m){
      boyfriend = m;
      }
      …
}
```

The two classes are meant to be used as shown in (c) below. Now see what happens if the two classes were used as in (d) below. Our code was not defensive enough to stop this "love tangle". In a situation like this, we say that the *referential integrity* has been violated. It simply means there is an inconsistency in object references.

```
Woman jean; Man james;                                   (c)
…
james.setGF(jean);  jean.setBF(james);
```

```
Woman jean; Man james, yong;                             (d)
…
james.setGF(jean);  jean.setBF(yong);
```



One way to prevent this situation is to implement the two classes as shown below:

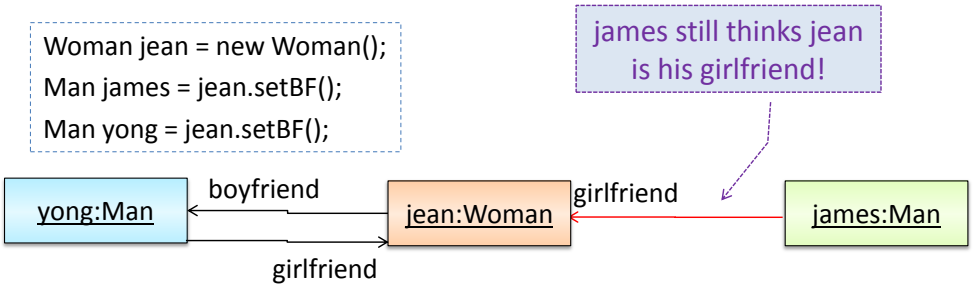
```
public class Woman {
  private Man boyfriend;
  public Man setBF() {
  boyfriend = new Man(this);
    return boyfriend;
  }
  …
}
```

Woman is in charge of setting/maintaining the link.

```
public class Man {
  private Woman girlfriend;
  Man(Woman w) {
        girlfriend = w;
  }
  …
}
```

```
Woman jean = new Woman();                                (e)
Man james = jean.setBF();
```

Now, the usage will be as given in (e). Note how the referential integrity is maintained. Also note that this code allows us to change the association later. That is, the association is *mutable* . If this association is indeed meant to be mutable, then we need to do additional checks to prevent a situation such as the one given below.



If the association is meant to be *immutable* (i.e. once created, it is not to be changed), we should write additional checks to preserve the immutability of the association. E.g.

```
if(boyfriend!=null) error("already has boyfriend");
```

As you can see, writing code that automatically preserves referential integrity requires a lot of additional work (on a related note, this is one good reason to avoid bidirectional associations).

The design-by-contract approach

Assume we are implementing an operation based on a precise operation behavior specified in the API (preconditions, post conditions, exceptions etc.). If we are following the defensive approach, our code should first check if the preconditions have been met. In contrast, the *Design-by-Contract* approach to coding assumes that it is the responsibility of the caller to ensure all preconditions are met. The operation will honor the contract only if the preconditions have been met. If any of them have not been met, the behavior of the operation is "unspecified".

DbC approach eliminates much of the extra checking required in defensive programming. Languages such as Eiffel have native support for DbC. For example, using Eiffel we can specify preconditions of an operation and the language runtime will check precondition violations without us having to do it in code. In languages such as Java and C++ where there is no built-in DbC support, DbC can only be effective if EVERYONE involved followed it consistently, which may be hard to achieve. It is safer to use defensive programming to protect your code.

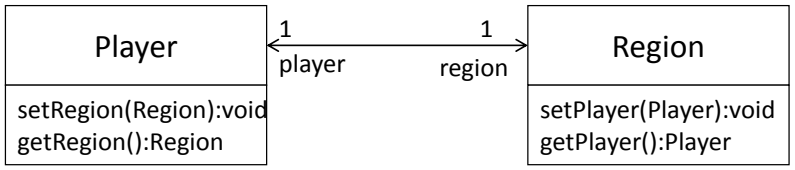
Worked examples

[Q1]

Imagine that we now support the following feature in our Minesweeper game.

> Feature id: *multiplayer*
>
> Description: a minefield is divided into mine regions. Each region is assigned to a single player. Players can swap regions. To win the game, all regions must be cleared.

Given below is an extract from our class diagram.



Minimally, this can be implemented like this.

```
class Player{
        Region region;
        void setRegion(Region r) { region = r;}
        Region getRegion() {return region;}
}
//Region class is similar
```

However, this is not very defensive. For example, a user of this class can pass a null to either of the methods, thus violating the multiplicity of the relationship.

Implement the two classes using a more defensive approach. Take note of the bidirectional link which requires us to preserve referential integrity at all times.

[A1]
In this solution, we assume Regions can be created without Players (note that we cannot be 100% defensive all the time). The usage will be something like this:

```
Region r1 = new Region();
Player p1 = new Player(r1);
Region r2 = new Region();
Player p2 = new Player(r2);
p1.setRegion(r2);
r1.setPlayer(p2);
```

Here are the two classes. Get methods are omitted as they are simple. Note how much extra effort we need to be defensive.

```
public class Region {

    private Player myPlayer;

    public Region(){
        //initialize region
    }

    public void setPlayer(Player newPlayer) {
        if (newPlayer == null) {
            stopSystemWithErrorMessage("Multiplicity violation");
        }
        if (myPlayer == newPlayer) {
            return; //same Player
        }
        if (myPlayer != null) {
            //I already have a Player!
            myPlayer.RemoveRegion(this);
        }

        myPlayer = newPlayer;
        //set the reverse link
        myPlayer.setRegion(this);
    }

    void RemovePlayer(Player disconnectingPlayer) {
        if(myPlayer == disconnectingPlayer)myPlayer = null;
        else stopSystemWithErrorMessage("unknown Player tring to disconnect");
    }

    private void stopSystemWithErrorMessage(String msg) {

    }
}
```

```
public class Player {

    private Region myRegion;

    public Player(Region region) {
        setRegion(region);
    }

    public void setRegion(Region newRegion) {
        if (newRegion == null) {
            stopSystemWithErrorMessage("Multiplicity violation");
        }
        if (myRegion == newRegion) {
            return;         //no change in Region!
        }
        if (myRegion != null) {
            //previous Region exists
            myRegion.RemovePlayer(this);
        }
        myRegion = newRegion;
            //set the reverse link
        myRegion.setPlayer(this);
    }

    public void RemoveRegion(Region disconnectingRegion) {
        if(myRegion == disconnectingRegion) myRegion = null;
    }

    private void stopSystemWithErrorMessage(String msg) {

    }

}
```

Note that the above code stops the system when the multiplicity is violated. Alternatively, we can throw an exception and let the caller handle the situation.

[Q2]
For the Manager class shown below, write an addAccount( ) method that

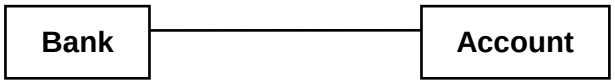
- restricts the maximum number of Accounts to 8
- avoids adding duplicate Accounts

[A2]
You may also use a non-generic class such as Vector instead of ArrayList generic class.

```
import java.util.*;
public class Manager {

   private ArrayList<Account> theAccounts ;

   public void addAccount(Account acc) throws Exception{
      if (theAccounts.size( ) == 8)
         throw new Exception ("adding more than 8 accounts");

      if (!theAccounts.contains(acc)) {
         theAccounts.add(acc) ;
      }
   }

   public void removeAccount(Account acc) {
      theAccounts.remove(acc) ;
   }
}//end class
```

[Q3]

Implement this bidirectional association. Note that the Bank uses accno attribute to uniquely identify an Account object. Assume the Bank class is responsibile for maintaining the links between objects.



[A3]

The code below contains a method in the Bank class to create an account; the bank field in the new account is thereby filled by the bank creating it.

We assume that once an Account has been assigned to one Bank, it cannot be assigned to a different Bank. Once the Account is removed from the Bank, it will not be used any more (hence, no need to remove the link from Account to Bank).

```
public class Account {
 private int accno ;
 private Bank theBank ;

 public Account(int n, Bank b) {
  accno = n ;
  theBank = b ;
 }
 public int getNumber() {
  return accno ;
 }
 public Bank getBank() {
  return theBank ;
 }
}//end class----------------------------------------
```

```
import java.util.*;

public class Bank {

   private HashMap<Integer,Account> theAccounts = new HashMap <Integer,Account> () ;

   public void createAccount(int n) {
      addAccount(new Account(n, this)) ;
   }
   public void addAccount(Account a) {
      theAccounts.put(a.getNumber(), a);
   }
   public void removeAccount(int accno) {
      theAccounts.remove(accno);
   }
   public Account lookupAccount(int accno) {
      return theAccounts.get(accno);
   }
}//end class
```

[Q4]

Implement the classes with appropriate references and operations to establish the association among the classes. Follow the defensive coding approach. Let the Marriage class handle setting/removal of reference.

[A4]

```
public class Marriage {
   private Man husband = null;
   private Woman wife = null;
   //extra information like date etc can be added

   public Marriage(Man m, Woman w) throws Exception {
      if (m==null || w==null)
         throw new Exception("no man/woman");

      if (m.isMarried() || w.isMarried())
         throw new Exception("already married");

      husband = m;
      m.enterMarriage(this);
      wife = w;
```

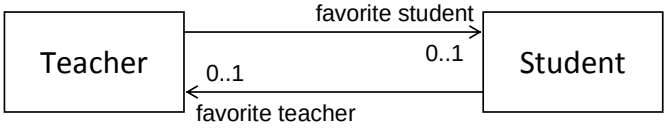
```
        w.enterMarriage(this);
      }

    public Man getHusband() throws Exception {
         if(husband == null) throw new Exception("error state");
        else return husband;
      }

      public Woman getWife() throws Exception {
         if(wife == null) throw new Exception("error state");
        else return wife;
      }

      // removal of both ends of  'Marriage'
      public void divorce() throws Exception {
        if (husband==null || wife==null) throw new Exception("no marriage");
        husband.removeFromMarriage(this);
        husband = null;
        wife.removeFromMarriage(this);
        wife = null;
      }
}//end class
```

[Q5]

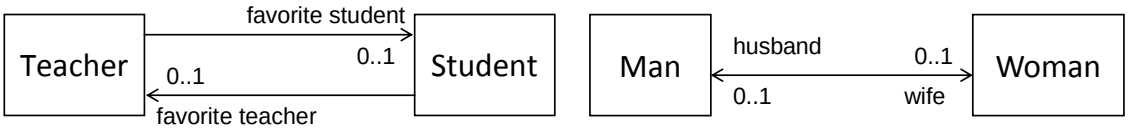
Is the code given below a defensive translation of the associations shown in the class diagram? Explain your answer.

```
class Teacher{
  private Student favStudent;
  void setFavStudent(Student s){
    favStudent = s;
  }
}
```

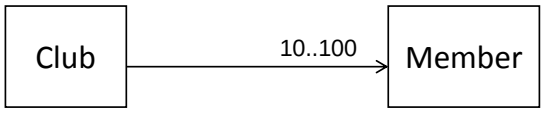
```
class Student{
  private Teacher favTeacher;
  void setFavTeacher(Teacher t){
    favTeacher = t;
  }
}
```



(b) In terms of maintaining referential integrity in the implementation, what is the difference between the following two diagrams?



(c) Show a defensive implementation of the remove(Member m) of the Club class given below.



[A5]
(a) Yes. Each links is mutable and unidirectional. A simple reference variable is suitable to hold the link.

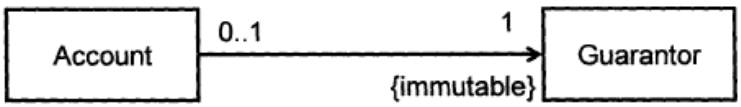
Teacher class can be made even more defensive by introducing a resetFavStudent() method unlink the current favorite student from a teacher. In that case, setFavStudent(Student) method should not accept null. This approach is more defensive because it prevents a null value being passed to setFavStudent(Student) by mistake and being interpreted as a request to delink the current favorite student from the Teacher object.

(b) First diagram has unidirectional links. Second has a bidirectional link. RI is only applicable to the second.

(c)

```
void removeMember(Member m){

        if (m==null) throw exception("this is null, not a member!");

        else if(member_count == 10) throw exception("we need at least 10 members to
        survive!");

        else if(!isMember(m)) throw exception ("this fellow is not a member of our club!");

        else members.remove(m); //members is a data structure such as ArrayList

}
```

[Q6]
Give a suitable defensive implementation to the Account class in the following class diagram. Note that "{immutable}" means once the association is formed, it cannot be changed.



[A6]

```
class Account{

        private Guarantor myGuarantor; //should not be public

        public Account(Guarantor g){

                if (g==null) haltWithErrorMessage("Account must have a guarantor");

                myGuarantor = g;

        }

        // there should not be a setGuarantor method

}
```

---End of document---