1. **Interfaces** The `Comparator` interface provides a way to control how a sort method (such as `Collections.sort`) sorts elements of a collection. For example, the following `main` method sorts a list of strings by their length by using a `StringLengthComparator` object:

```
public static void main(String[] args) {
    List<String> t = new ArrayList<>();
    t.add("a very very very very long string");
    t.add("a short string");
    t.add("a medium length string");
    Collections.sort(t, new StringLengthComparator());
    System.out.println(t);
}
```

The `Comparator` interface is defined as follows:

```
public interface Comparator<T> {
    /**
     * Compares its two arguments for order. Returns a negative
     * integer, zero, or a positive integer as the first argument
     * is less than, equal to, or greater than the second.
     *
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, zero, or a positive integer as
     * the first argument is less than, equal to, or greater than
     * the second.
    public int compare(T o1, T o2);

    // ...
}
```

Implement the class `StringLengthComparator` so that strings are sorted by their length:

```
public class StringLengthComparator implements Comparator<String> {

    public int compare(String s, String t) {



    }
}
```

Solution:

```
public class StringLengthComparator implements Comparator<String> {

    public int compare(String s, String t) {
        return Integer.compare(s.length(), t.length());
    }
}
```

Modify your implementation so that `StringLengthComparator` first sorts by string length then by dictionary order (i.e., if two strings have the same length then they are sorted by dictionary order):

```
public class StringLengthComparator implements Comparator<String> {

    public int compare(String s, String t) {




    }
}
```

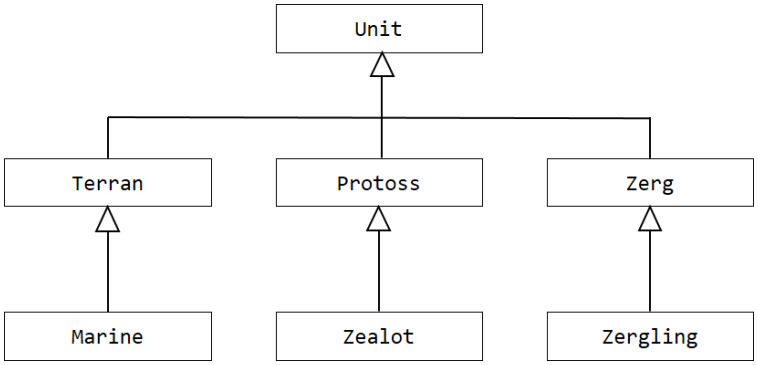
Solution:

```
public class StringLengthComparator implements Comparator<String> {

    public int compare(String s, String t) {
        int result = Integer.compare(s.length(), t.length());
        if (result == 0) {
            result = s.compareTo(t);
        }
        return result;
    }
}
```

2. **Inheritance terminology**

A simplified inheritance hierarchy for the video game Starcraft is shown below:



(a) Unit is a superclass of Terran

(b) Protoss is a subclass of Unit

(c) Zergling is a subclass of Zerg

(d) Marine is not related to of Protoss

(e) Object is an ancestor of Unit

3. **Composition instead of inheritance**

Implement `Stack` using composition instead of inheritance.

```
public class Stack {
    private List<Integer> stack;

    public Stack() {



    }

    public Stack(Stack other) {



    }

    public void push(int value) {



    }

    public int pop() {




    }
}
```

Solution:

```
public class Stack {
    private List<Integer> stack;

    public Stack() {
        this.stack = new ArrayList<>();
    }

    public Stack(Stack other) {
            this.stack = new ArrayList<>(other.stack);
    }

    public void push(int value) {
```

```
            this.stack.add(value);
    }

    public int pop() {
        if (this.stack.isEmpty()) {
            throw new NoSuchElementException();
        }
        return this.stack.remove(this.size() - 1);
    }
}
```

4. **Subclass constructors**

   Refer back to the figure in Question 2. Every `Unit` has an amount of health; suppose that the constructors for `Unit` are implemented like so:

```
public class Unit {
    private int health;

    public Unit() {
        this.health = 1;
    }

    public Unit(int health) {
        this.health = health;
    }

}
```

   (a) Implement the `Terran` constructors; remember that `Terran` does not have direct access to the field `health`:

```
public class Terran extends Unit {

    public Terran() {


    }

    public Terran(int health) {


    }

}
```

Solution:

```
public class Terran extends Unit {

    public Terran() {
```

```
        super();
    }

    public Terran(int health) {
        super(health);
    }

}
```

(b) In addition to health, every `Protoss` unit has an amount of shields. Implement the `Protoss` constructors:

```
public class Protoss extends Unit {
    private int shields;

    // initialize unit to 1 health and 1 shields
    public Protoss() {



    }

    public Protoss(int health, int shields) {



    }

}
```

Solution:

```
public class Protoss extends Unit {
    private int shields;

    // initialize unit to 1 health and 1 shields
    public Protoss() {
        super();
        this.shields = 1;
    }

    public Protoss(int health, int shields) {
        super(health);
        this.shields = shields;
    }

}
```

5. Suppose you have a class Y that extends X. X has a method with the following precondition:

   `@pre. value must be a multiple of 2`

   If Y overrides the method which of the following are acceptable preconditions for the overriding method? Provide a brief statement explaining your answer for each precondition.

   (a) `@pre. value must be a multiple of 2`

   > **Solution:** Acceptable because the precondition is the same as in the superclass.

   (b) `@pre. value must be odd`

   > **Solution:** Not acceptable because the precondition of the overriding method does not satisfy the precondition of the parent method.

   (c) `@pre. value must be a multiple of 2 and must be less than 100`

   > **Solution:** Not acceptable because the precondition of the overriding method does not allow for values greater than or equal to 100 whereas the parent method has not such restriction.

   (d) `@pre. value must be a multiple of 10`

   > **Solution:** Not acceptable because the precondition of the overriding method does not allow for all values that are multiples of 2.

   (e) `@pre. none`

   > **Solution:** Acceptable because every value that satisfies the parent method precondition also satisfies the precondition of the overriding method.

6. Suppose you have a class Y that extends X. X has a method with the following postcondition:

   `@return a string of length 10`

   If Y overrides the method which of the following are acceptable postconditions for the overriding method? Provide a brief statement explaining your answer for each postcondition.

   (a) `@return a string of length at least equal to 10`

   > **Solution:** Not acceptable because strings with length not equal to 10 do not satisfy the postcondition of the parent method.

   (b) `@return the string equal to "weimaraner"`

   > **Solution:** Acceptable because `"weimaraner"` is a string of length 10.

   (c) `@return the empty string`

   > **Solution:** Not acceptable because the string with length 0 does not satisfy the postcondition of the parent method.

   (d) `@return a string of length 10`

   > **Solution:** Acceptable because the postcondition is identical to that of the parent method.

   (e) `@return a random string of length 10`

   > **Solution:** Acceptable because a random string of length 10 is a string of length 10.

7. In the `Mix` class from the lecture slides, which of the following are legal exception specifications? Provide a brief statement explaining your answer for each method header.

```
@Override
public void someDogMethod() throws BadDogException
```

   > **Solution:** Acceptable because `BadDogException` is substitutable (is a subclass) for `DogException`

   (a)
```
@Override
public void someDogMethod() throws Exception
```

   > **Solution:** Not acceptable because `Exception` is not substitutable for `DogException` (`Exception` is an ancestor of `DogException`)

   (b)
```
@Override
public void someDogMethod()
```

Solution: Acceptable because no exception is thrown.

(c)
```
@Override
public void someDogMethod() throws DogException, IllegalArgumentException
```

Solution: Trick question. `IllegalArgumentException` is an unchecked exception and including it in the method header doesn't actually do anything useful. The Java compiler will allow you to write code such as this, so this is technically legal. From a substitutability point of view, this is not acceptable because the overriding method is stating that it can fail (throw an `IllegalArgumentException`) in situations that the the parent method does not.