Instuderingsuppgifter läsvecka 5 - LÖSNINGAR

1.

```
public class NewSetAdapter<T>  implements Set<T>{
   private NewSet<T> ns = new NewSet<T>();
   public void add(T e) {
      ns.insert(e);
   }
   public void delete(T e) {
      ns.remove(e);
   }
   public int size() {
      return ns.size();
   }
   public boolean has(T e) {
      return ns.contains (e);
   }
}//NewSetAdapter
```

2.

a) Syftet med designmönstret Decorator är att tillhandahålla ett sätt att dynamiskt lägga till nya beteenden och tillstånd till ett objekt.

b)

```
public interface IA {
   public void f();
}//IA
```

```
public class A implements IA {
   public void f() {
      System.out.println("A");
   }
}//A
```

```
public class B implements IA {
   private IA ia;
   public B(IA ia) {
      this.ia = ia;
   }
   public void f() {
      ia.f();
      System.out.println("B");
   }
}//B
```

```
public class C implements IA {
   private IA ia;
   public C(IA ia) {
      this.ia = ia;
   }
   public void f() {
      ia.f();
      System.out.println("C");
   }
}//C
```

```
public static void main(String[] args) {
   IA a = new A();
   IA b = new B(new A());
   IA c = new C(new A());
   IA bc = new C(new B(new A()));
   IA cb = new B(new C(new A()));
   a.f();
   b.f();
   c.f();
   bc.f();
   cb.f();
}//main
```

3.

```
public class A extends Observable {
   private int value = 0;
   public void compute(int x) {
      value += x;
      setChanged();
      notifyObservers();
   }
   public int getValue() {
      return value;
   }
}//Observable

public class B implements Observer {
   private A theAObject;
   public B(A anAObject) {
      theAObject = anAObject;
      theAObject.addObserver(this);
   }
   public void update(Observable o, Object arg) {
      if (o instanceof A) {
         System.out.println(theAObject.getValue());
      }
   }
}//Observer
```

4.

Syftet med MVC-mönstret är att frikoppla användargränsittet från den underliggande datamodellen och därigenom möjliggöra att informationen i datamodellen kan presenteras på olika sätt utan att datamodeller behöver förändras.

I MVC-mönstret delas applikationen upp i tre komponenter Model, View och Control:

- Model utgör den domänspecifika representationen av den informationen som applikationen bearbetar. Utgörs av de klasser som ansvarar för tillståndet i applikationen.
- View är de klasser som ansvarar för att presentera modellen för användaren (normalt ett grafiskt användargränssnitt).
- Control är de klasser som handhar de händelser som påverkar modellens tillstånd och styr interaktionen mellan Model och View.

5.

Syftet med designmönstret *Façade* är att tillhandahålla ett förenklat och enhetligt gränssnitt till ett komplext delsystem bestående av en mängd klasser/gränsnitt. Användarna ser bara fasaden och blir således inte beroende av delsystemets interna komplexitet. Vilket i sin tur betyder att den interna strukturen i delsystemet kan ersättas med en annan, förhoppningsvis enklare design, utan att användarna påverkas.

6.

Ett kontrollerande (*checked*) undantag *måste hanteras* i programmet, antingen genom att det fånga i metoden där det inträffar eller genom att i händelselista för denna metod ange att det kastas vidare till den anropande metoden.

När en metod kastar ett icke-kontrollerand (*unchecked*) undantag behöver inte detta anges i metodens händelselista.

Okontrollerade undantag används för att signalera programmeringsfel, dvs. buggar som uppstår p.g.a. att programmeraren inte gjort sitt jobb korrekt. Okontrollerade undantag skall (normalt) inte fångas. Ju förr det smäller (dvs felet uppstår) desto bättre. Programmeraren blir medveten om buggen och kan åtgärda den.

Kontrollerande undantag används för att signalera fel som ligger utanför programmerans kontroll, t.ex. att man försöker läsa en fil som inte finns eller att den ljudenhet som man försöker använda inte är tillgänglig. Detta är fel som inträffar vid interaktionen med programmet och som är möjliga att hantera. Om t.ex. en fil inte finns skall inte programmet terminera, användaren har kanske uppgivit felaktigt namn på filen. Istället skall det uppkommna kontrollerade untantaget fångas och användaren ges möjligheten att uppge rätt filnamn.

7.

try-blocket	innesluter den kod i vilken undantag kan kastas.
catch-blocket	anger vilken typ av undantag som skall fångas och innesluter den kod som skall utföras om den angivna typen av undantag inträffar i **try**-blocket.
finnaly-blocket	innesluter den kod som skall utföras oberoende av om ett undantag inträffar eller inte.

8.

```
public class Maximum {
   public static void main(String[] args) {
      if (args.length >= 2 ) {
         try {
            int t1 = Integer.parseInt(args[0]);
            int t2 = Integer.parseInt(args[1]);
            int max = t1 + t2;
            System.out.println("The maximum of " + t1 + " and " + t2 + " is " + max);
         }
         catch (NumberFormatException e) {
            System.out.println("Invalid input values " + e.getMessage());
            System.out.println("The input values must be integers.");
         }
      }
      else
         System.out.println("Usage: java Maximum interger1 integer2");
   }//main
}//Maximum
```

9.

Alternativ d) är felaktig, eftersom undantagsklassen FirstException är en superklass till undantagsklassen SecondException.

10.

a) och b) är korrekta, medan c) är felaktig. En metod som överskuggar en metod får inte kasta ett kontrollerande undantag om inte metoden som överskuggas gör det. Om detta vore tillåtet skulle den överskuggade metoden i subklassen ställa större krav på klienten än vad metoden gör i superklassen (eller som i detta fall i interfacet), eftersom klienten måste ta hand om undantaget genom att antingen fånga det eller skicka det vidare.