Lösningsförslag: Övning 3.

Uppgift 1

Metoden strider mot designprincipen *Separation of Concern*. Enligt denna principen skall en metod endast göra en sak och göra denna bra. Metoden changeTemp gör tre saker:

1) kontrollerar för varje lägenhet om temperaturen behöver justeras
2) tar reda på hur mycket temperaturen behöver justeras
3) justerar temperaturen

För att åstadkomma att metoden changeTemp följer *Separation of Concern* utför vi en refaktorering i vilken vi *successivt* extraherar metoder. Genom att följa *Separation of Concern* får man små och tydliga metoder som både är lätta att läsa och förstå.

```
public void changeTemp() {
   for (Apartment a : apartments) {
      changeTempIfNeeded(a);
   }
}
```

```
private void changeTempIfNeeded(Apartment a) {
   if (!a.tempIsPerfect())
      setTemp(a);
}
```

```
private void setTemp(Apartment a) {
   int change = a.tempDifference();
   a.setTemp(change);
}
```

Vi kan också konstatera att metoden strider mot *Information Expert Pattern* och principen *Tell, don't ask*. Metoden

```
public void changeTemp() {
   for (Apartment a : apartments) {
      if (!a.tempIsPerfect()) {
         int change = a.tempDifference();
         a.setTemp(change);
      }
   }
}
```

frågar objektet (som refereras via variabeln a) om dess tillstånd, utvärderar detta tillståndet och utifrån detta ändrar tillståndet hos objektet. Om vi har ansvaret för klassen Apartment borde också denna klass förändras. Följs *Information Expert Pattern* och principen *Tell, don't ask*, får metoden changeTemp följande utseende:

```
public void changeTemp() {
   for (Apartment a : apartments) {
      a.changeTempIfNeeded();
   }
}
```

och i klassen Apartment införs metoden changeTempIfNeeded:

```
public void changeTempIfNeeded() {
   if (!tempIsPerfect()) {
      int change = tempDifference();
      setTemp(change);
   }
}
```

Uppgift 2

a)
```
/**
   @pre   a != null and 0 < n <= a.length
   @post  none
   @returns  returns the mean of the first n numbers in a
**/
```

b) Ett uppenbart exempel är när str är ”tom”, dvs innehåller noll tecken. Ett annat exempel är t.ex. strängen ”ab”. Eller mer generellt, varje sträng där en eller flera tecken förekommer med samma frekvens, så är “most frequently” inte tillräckligt definierat.

Uppgift 3

a) Förvillkoret till metoden setTargetSpeed är omöjligt för användaren att kontrollera eftersom isActive är en privat instansvariabel. Genom att införa en accessmetod isActive() kan detta problem rättas till.

```
/**
* Checks whether the cruise control is active or not
*/
public boolean isActive() {
   return isActive;
}
```

Specifikationen blir då:

```
/**
 * Changes the preferred speed which the cruise control tries to achieve
 * @param      speed the new preferred speed
 * @pre        speed >= 0 && isActive()
 * @post        isLegal returns true
 */
```

Det är också rimligt att använda en assertion högst upp i implementationen av setTargetSpeed för att kontrollera att förvillkoret är uppfyllt innan metoden körs. Det underlättar verifikations- och valideringsprocessen.

```
public void setTargetSpeed(float speed) {
   assert speed >= 0 && isActive;
   assert invariant();
   if (isLegal(speed))
       targetSpeed = speed;
   else
       targetSpeed = legalSpeed;
   assert invariant();
}
```

b) Nej, klassinvarianten måste vara uppfylld då objektet befinner sig i ett stabilt tillstånd (d.v.s. när inga metoder eller konstruktorn för närvarande körs på objektet). Alltså måste den vara uppfylld efter att konstruktorn exekverat samt alltid då en metod har exekverat.

I konstruktorn samt metoden setTargetSpeed har programmeraren tänkt till och gjort sitt jobb och klassinvarianten gäller efter exekvering. Däremot tänkte han/hon/hen nog för mycket på fikarast och för lite på klassinvariant när metoden activate skrevs. Här sätts hastigheten till ett värde som klienten skickar in, utan att värdet kontrolleras.

En lösning på detta problem skulle kunna vara att i metoden activate använda sig av ett anrop till setTargetSpeed istället:

```
/**
* Activates the Cruise control
* @pre   speed >= 0
* @post targetSpeed >=0
*/
public void activate(float speed) {
   assert invariant();
   assert speed >= 0;
   isActive = true;
   setTargetSpeed(speed);
   assert invariant();
}
```

(Observera att det egentligen inte är lämpligt att metoden activate både aktiverar och ställer in målvärde för farten. Ett mer logiskt beteende skulle vara att den bara aktiverar objektet)

Uppgift 4

a) II är starkare, eftersom förvillkoret är svagare och eftervillkoret är starkare.

b) III är starkare, eftersom eftervillkoret är starkare.

c) IV är starkare, eftersom förvillkoret är svagare och eftervillkoret är starkare.

d) II är starkare, eftersom förvillkoret är svagare.

e) Specifikationerna är inte kompatibla. Om x <= 0 kastar specifikation II IllegalArgumentException, medan specifikation IV returnerar Double.NEGATIVE_INFINITY.

f) IV är starkare, eftersom förvillkoret är svagare.

Kommentarer:

Att jämföra två specifikationer för att avgöra vilken av specifikationerna som är starkast förutsätter givetvis att specifikationerna är kompatibla (dvs båda specifikationerna måste tillhandahålla samma grundfunktionallitet).

Ju starkare en specifikation är, ju svårare är den att implementera.

Ju svagare förvillkor, ju starkare specifikation.

Ju starkare eftervillkor, ju starkare specifikation.

Då två specifikationer jämförs och den ena specifikationen har ett svagare förvillkor och den andra specifikationen har ett starkare eftervillkor går det inte att avgöra vilken av dessa två specifikation som är starkast.

Uppgift 5

a) Tillåtet i Java. En äkta subtyp enligt LSP, eftersom en subtyp får ha starkare eftervillkor.

b) Tillåtet i Java. Ingen äkta subtyp enligt LSP, eftersom en subtyp inte får ha starkare förvillkor.

c) Ej tillåtet i Java och ingen äkta subtyp enligt LSP, eftersom beteendet förändras när en checked exception kastas.

d) Tillåtet i Java. Ingen äkta subtyp enligt LSP, eftersom CTH har en svagare specifikation.

e) Tillåtet i Java (kovarians). En äkta subtyp enligt LSP.

Kommentarer:

Enligt Liskovs substitutionsprincip måste en subklass uppträda (allra minst) som dess superklass. Detta innebär att en subklass inte får ta bort något, men däremot lägga till.

Överskuggning uppstår mellan två eller flera metoder med samma namn och samma parameterlistor i klasser med arvsförhållande.

Som generell minnesregel: Vid överskuggning agerar metoden i subklassen i stället för metoden i superklassen, och därför måste metoden i subklassen *uppfylla kontraktet för metoden i superklassen* (den ska uppträda minst "lika bra").

I Java måste följande vara uppfyllt vid överskuggning:

- Synligheten för metod i subklassen måste vara minst lika stor som synligheten för metoden i superklassen.
- Metoden i subklassen får inte kasta några checked exceptions som inte metoden i superklassen gör (men den behöver inte kasta dem! Metoden i subklassen får alltså inte göra "mer fel" än metoden i superklassen).
- Returtypen för metoden i subklassen måste vara utbytbar mot returtyp för metoden i superklassen.

Då en klass Sub ärver av en klass Super och någon metod omdefinieras (kallas *overriding* (överskuggning) för instansmetoder och *hiding* (döljande) för klassmetoder, krävs att returtyperna är utbytbara (return type-substitutable). Om returtypen till metoden i subklass Sub är r_{sub} och returtypen till superklassen Super är r_{super}, måste antingen $r_{sub} = r_{super}$ eller rs_{ub} vara en subtyp till r_{super}.

Uppgift 6

Nedanstående kod är korrekt enligt Javas typregler och går således att kompilera, men ett exekveringsfel uppstår när ett objekt av typen Double lagras i ett fält av typen Integer:

```
Integer[] ia = new Integer[10];
Number[] na = ia;
na[0] = new Double(3.14); //går även att ange na[0] = 3.14; (autoboxing)
```