1. Mi a baj az alábbi osztálydefinícióval? (1 pont)

```
class A {
    private int v;
    public static int getV(){ return v; }
}
```

Osztályszintű metódusból példányszintű attribútumra hivatkozni minősítés nélkül nem lehet, fordítási hibát kapunk. A v-re történő hivatkotás, ha nem lokális változó vagy formális paraméter, akkor this.v-nek értelmezhető, de osztályszintű metódusban nincs this paraméter.

2. Ismertesd a felüldefiniálás és túlterhelés fogalmát, különbségüket, a rájuk vonatkozó szabályokat! (2 pont)

Metódusok felüldefiniálhatók és túlterhelhetők. Egy típusdefinícióban bevezethetünk (illetve bázistípusokból megörökölhetünk) metódusokat, melyeknek ugyanaz a neve, de különböző a paraméterezése: a formális paraméterek számában vagy deklarált típusában eltérnek; úgy is mondhatjuk, hogy más a szignatúrájuk. Ezt túlterhelésnek nevezzük. Az is ebbe a konstrukcióba sorolható, ha egy metódust megöröklünk, és mellé ugyanolyan nevű, de különböző szignatúrájú metódust vezetünk be egy típusdefinícióba. Ilyen esetekben a metódus meghívásakor az aktuális paraméterek deklarált (statikus) típusából egyértelműen meghatározható kell legyen, hogy a túlterhelt változatok melyikét kívánjuk meghívni. Ha nem dönthető el egyértelműen a hívás célja, fordítási hibát kapunk. (Ehhez kapcsolódóan bevezeti a Java a "jobban illeszkedő hívás" fogalmát: ha altípusokban térnek el a túlterhelt változatok, akkor egy hívás több definícióra is illeszkedhet; ha ilyenkor létezik az adott szituációban "legjobban illeszkedő", akkor azt választja a fordító, egyébként fordítási hibát ad.

A felüldefiniálás hasonlóan néz(het) ki, mint a túlterhelés, de ebben azt esetben a szignatúra nem változhat: ugyanaz a paraméterezése mindegyik definíciónak. Ebben az esetben egy őstípusból megörökölt példánymetódus (soha nem osztályszintű: static esetben nem beszélhetünk felüldefiniálásról) implementációját cserélhetjük le a leszármazott osztályban. Ilyenkor nem új metódusról, csak a meglévő metódus újabb implementációjáról beszélünk. A különböző implementációk közül futási időben választja ki a rendszer a legjobban illeszkedőt, azaz azon objektum, amelyre meghívtuk, dinamikus típusához legközelebbi ős (önmagát is beleértve) változatát. A felüldefiniált változatok esetében a szignatúra ezek szerint megegyezik, de a láthatósága a műveletnek bővülhet, a visszaadott érték típusa (ha nem void vagy primitív) szűkülhet (azaz lehet az ősbeli implementációban deklarált visszatérési típus egy altípusa), és a metódus throws listája is változhat, de csak olyan módon, hogy olyan ellenőrzött kivételt, amit a megörökölt implementáció nem deklarált, az új implementáció sem deklarálhat. Ezek a szabályok nem sértik a helyettesítési elvet.

Összefoglalva: a túlterhelt metódusok eléggé különböznek ahhoz, hogy fordítási időben, a statikus típusok alapján válasszon a fordító közülük a hívásnál, a felüldefiniált implementációk pedig eléggé hasonlítanak ahhoz, hogy futás közben kelljen választani a hívott objektum dinamikus típusa alapján.

3. Hogyan működik és mire való a finally a try utasításban? (2 pont)

Egy try utasítás tartalmazhat (legfeljebb egy) finally blokkot (az utasítás legvégén), mely mindenféleképpen megkapja meg a vezérlést, ha a try blokkba belépett a vezérlés. Ha a try blokk normálisan lefutott, megkezdődik a finally végrehajtása. Ha a try blokk kivételt váltott ki, és nem volt (vagy nem volt megfelelő, azaz a kivétel objektum dinamikus típusára illeszkedő) catch ág, akkor a kivétel terjedése előtt is megkíséreljük a finally végrehajtását. Ha a try blokkban fellépő kivétel lekezelésére van a try utasításban catch ág, akkor először az hajtódik végre: akár sikeresen, akár (egy újabb, vagy a try blokkban fellépett) kivétellel fejeződik be, a finally végrehajtása megkezdődik. Elképzelhető, hogy a finally is kivételt vált ki, azaz nem fut le teljes egészében – ilyenkor a try blokkban vagy a catch ágban fellépett kivétel helyett ez az új kivétel terjed tovább (kivéve a try-with-resources utasítás esetét, melyben az implicit finally blokk

megőrzi a korábban fellépett kivételt, és csak utána láncolja az újabbat).

A finally szerepe az erőforráskezelésben van: ha Java objektumokkal olyan erőforrásokat reprezentálunk, amelyek felszabadítását explicit el kell végezni (nem pusztán a Java virtuális gép által kezelt memória, hanem például egy filehandler), akkor ezt az explicit felszabadítást igyekszünk minél korábban elvégezni (hogy ne tartsuk lekötve az erőforrást feleslegesen, ezzel segítve a többi folyamat, szál, program haladását). Ahelyett, hogy az erőforrás felszabadítását az objektum élettartamához kötnénk, ami a szemétgyűjtés miatt akár túl hosszúra is nyúlhat, egyből az erőforrás használata után felszabadítjuk – jellemzően egy close metódus szokott erre lenni. Azt, hogy a felszabadítás mindenféleképpen (fellépő kivételek esetén is) megtörténjen, vagy legalábbis megkíséreljük azt, az erőforrás használatát try-finally szerkezetbe zárjuk:
Resource r = …; try { … using resource … } finally { r.close(); }
Több erőforrás használatánál egymásba is ágyazódhatnak az ilyen szerkezetek. Más idiómák is léteznek az erőforráskezelés leírására, illetve a finally szerkezet használatára.
AutoCloseable interfészt megvalósító osztállyal reprezentált erőforrások esetén elegánsabb a Java 7-től rendelkezésre álló try-with-resources utasítás használata.

4. Mi a statikus és a dinamikus típus szerepe, viszonya? Mi a különbség közöttük? Melyik mihez szükséges? (2 pont)

A változók (beleértve a formális paramétereket is) esetén a változó deklarációjában szereplő típust statikus típusnak nevezzük. Ha a változó referenciatípusú és nem null értékű, akkor az általa hivatkozott objektum létrehozásához használt osztályt pedig dinamikus típusnak hívjuk. A Java nyelv szabályai garantálják, hogy a dinamikus típus altípusa a statikus típusnak. Például ilyen szabály a következő: egy értékadás típushelyes, ha a baloldalán álló kifejezés statikus típusának altípusa a jobboldalon álló kifejezés statikus típusa. Sokszor a dinamikus típus megegyezik a statikussal, de van, amikor valódi altípusa annak. Statikus típus lehet absztrakt osztály vagy interfész is, dinamikus típus azonban csak konkrét osztály lehet.

A statikus (deklarált) típusok a fordító által használt típusok: ezek alapján dönti el a fordító, hogy a program típushelyes-e, és hogy le lehet-e fordítani azt. A lefordított program futtatásánál játszanak szerepet a dinamikus típusok: egy példánymetódus meghívásánál a példány dinamikus típusa alapján dől el, hogy a metódus melyik implementációja hajtódjon végre (felüldefiniált metódus esetén). Használjuk a dinamikus típust dinamikus típusellenőrzésnél is (type cast és instanceof).

A statikus típus tehát már fordítási időben, static time, ismert a fordító számára, a dinamikus típus csak futási időben (dynamic time) határozható meg általános esetben, ezért a fordítási idejű típusellenőrzés során nem vesszük figyelembe (de figyelembe vehetjük dinamikus típusellenőrzéshez).

Egy változó statikus típusa állandó, a dinamikus típusa azonban értékadásról értékadásra változhat. Ezért nem lehet általános esetben eldönteni fordítási időben, hogy egy változónak a program egy adott pontján mi a dinamikus típusa: többféle is lehet neki.

A dinamikus típus speciálisabb (lehet), mint a statikus, ezért “pontosabbnak” tekintjük. Ez indokolja a statikus típusrendszer kiegészítését dinamikus típusellenőrzéssel: így olyan rugalmassághoz jutunk, amely mellett jó viselkedésű, de nem helyes (nem fordítható) programokat is helyessé tehetünk type castok bevezetésével.

5. Mi a különbség az absztrakt osztályok és az interfészek között? (2 pont)

Az absztrakt osztályok többet tudnak valamilyen szempontból, mint az interfészek: tartalmazhatnak nem public, nem static és nem final mezőket is, valamint tartalmazhatnak implementációval rendelkező, tehát nem abstract metódusokat is.

Más szempontból az interfészek is többet tudnak az absztrakt osztályoknál: részt vehetnek többszörös öröklődésben. Azaz az interfészek között megengedett a többszörös öröklődés, és egy osztály is implementálhat több interfészt is.

6. Mit jelent a bounded (korlátos) a bounded parametric polymorphism (korlátos parametrikus polimorfizmus) kifejezésben? (1 pont)

A korlátos parametrikus polimorfizmus esetén egy generikus (típus- vagy metódus-) definíció formális típusparaméterére (paramétereire) megkötés(eke)t tehetünk: ilyen formális típusparaméternek csak olyan típust feleltethetünk meg aktuálisként, mely a megkötés(eke)t teljesíti. Például <T extends Shape>.

Kétféle megkötés (bound) létezik Javában: upper bound és lower bound. Az upper bound esetén az öröklődési hierarchiában felülről korlátozzuk az aktuálisként számításba vehető típusok körét, azaz egy elvárt bázistípust köthetünk ki: <T extends Shape>. A lower bound esetén alulról korlátozunk, azaz egy típus elvárt altípusát kötjük ki. Ezt a korlátot csak a wildcard (?) kapcsán alkalmazhatjuk: <? super Rectangle>.

7. Hogyan néz ki az altípus reláció a referenciatípusokon? Magyarázd el a három összetevőjét, és a rájuk vonatkozó szabályokat! Milyen gráfot határoz meg a reláció? (2 pont)

Azon túl, hogy a primitív típusok között definiál altípus relációt (automatikus típuskonverziót) a Java, mi szerint: b < s|c < i < l < f < d, a referencia típusok felett is bevezeti az altípus relációt, mely egy részbenrendezés: reflexív, antiszimmetrikus és tranzitív. Így a rendezés gráfja egy irányított, körmentes gráf. A reláció három másik reláció uniójának reflexív tranzitív lezártja: az osztályöröklődés, az interfészöröklődés és a megvalósítás relációké.

Az osztályöröklődés irányított fa, melynek gyökere a lava.lang.Object, éleit pedig a programban a bázisosztályt megadó extends (beleértve az implicit extends java.lang.Object) határozza meg. Az interfészöröklődés az interfészek közötti extends reláció, mely többszörös is lehet, ezért gráfja egy irányított körmentes gráf. A megvalósítás reláció interfészekből osztályokba mutató éleket jelent az implements kulcsszó használatának megfelelően: gráfja irányított körmentes.

8. Írd fel a lusta és a mohó konjunkció ("és") művelet művelettábláját! Magyarázd el a különbséget a kettő között! (2 pont)

mohó (&)

	I	H	K	V
I	I	H	K	V
H	H	H	K	V
K	K	K	K	K
V	V	V	V	V

lusta (&&)

	I	H	K	V
I	I	H	K	V
H	H	H	H	H
K	K	K	K	K
V	V	V	V	V

Ha a konjunkció baoldalán szereplő kifejezés hamisra értékelődik ki, a lusta operátor nem értékeli ki a jobb operandusát, az eredmény automatikusan hamis lesz. A mohó operátor kiértékeli ebben az esetben is a jobb operandust, így az eredmény lehet kivétel vagy végtelen számítás is (a a jobb operandus az).

9. Mi a szerepe az operátorok asszociativitásának a kifejezéskiértékelésben? (1 pont)

Azonos precedenciaszintű operátorok használata esetén határozza meg a kiértékelési sorrendet. Például A + B - C kiértékelése során először A, majd B értékelődik ki, ezután az összeadás, majd C, és végül a kivonás – mert az összeadás és a kivonás balasszociatív, azaz balról zárójelezendő. Az értékadás jobbasszociatív.

10. Rajzold le, hogy a "java C alma barack" programfuttatás során az m nevű metódus befejeződése előtti pillanatban mi található a stacken és a heapen! (2 pont)

```
public class C {
  String str;
  int num;

  public C(String str, int num) {
    this.str = str;
    this.num = num;
  }

  public void m( int i ){
      num = i;
      i = str.length();
  }

  public static void main( String[] args ) {
    String str = args[1];
    int num = str.length();
    C obj = new C(str, num);
    obj.m(num/2);
  }
}
```