Tehtävä 1

A.

ArrayListin periytymishierarkia:

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

B.

LinkedListin periytymishierarkia:

```
 java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.AbstractSequentialList<E>
                java.util.LinkedList<E>
```

C.

Jos ArrayList ja LinkedList periytyisivät suoraan Object-luokasta, tulisi molempien luokkien sisältää kaikki mahdollinen toiminnallisuus niiden toteuttamiseksi. Tämä johtaisi väistämättä siihen, että molempien luokkien määrittämiseen tarvittava koodi olisi huomattavasti pidempi. Molempien luokkien ja Object-luokan väliin jäävät luokat pitäisi myös kijroittaa, siten että ne sisältäisivät kaiken niiden toiminnallisuuteen tarvittavan koodin. Kun hyödynnyten periytyvyyttä, voidaan jo kirjoitettua koodia käyttää uudelleen, koska hierarkiassa alemmat luokat perivät ylemmän luokan metodit ja ylemmän luokan sisäiset muuttujat. Metodeja voidaan myös mahdollisesti määritellä uudelleen, jotta voidaan hyödyntää polymorfismia. Lisäksi hierarkian avulla on huomattavasti helpompi määritellä ja dokumentoida esimerkiksi kirjastojen dokumentaatioita, koska hierarkian perusteella ilmenee eri luokkien väliset suhteet selkeästi.

D.

ArrayList: RandomAccess

Rajapinta, jota esimerkiksi List-toteutukset käyttävät osoittaakseen, että ne tukevat nopeaa, usein vakioaikaista, satunnaiskäyttöä. Tämän rajapinnan ensisijainen tarkoitus on antaa geneeristen algoritmien muuttaa käyttäytymistään niin, että ne tarjoavat hyvän suorituskyvyn, kun niitä sovelletaan joko "sequential accessin" tai "random accessin" kanssa.

LinkedList: Deque

Tämä rajapinta mahdollistaa jono-tyylisten tietorakenteiden muodostamisen. Jonot ovat lineaarisia kokoelmia, jotka tukevat alkioiden lisäämistä ja poistamista molemmista päistä. Rajapinta sisältää metodeja alkion lisäämistä, poistamista ja tarkastelua varten. Kukin näistä metodeista on olemassa kahdessa muodossa: toinen heittää poikkeuksen, jos operaatio epäonnistuu, ja toinen palauttaa erikoisarvon (joko nolla tai false, operaatiosta riippuen).

E.

Rajapintojen avulla voidaan saavuttaa täydellinen abstraktio, kun voidaan kirjoittaa vain rutiinien määrittelyt, mutta niiden toiminnallisuutta ei tarvitse kirjoitta vielä. Esimerkiksi ArrayListin ja LinkedListin tapauksessa tarkoituksena on käyttää perintämekanismia siten, että rajapinnan toteuttamien luokkien taataan osaavan määrittelyn mukaisen operaation ja luokan kirjoittajalta voidaan varmistaa, että näitä operaatioita halutaan todellakin käytettävän.

F.

Kokeillaan konstruktorin toimintaa käytännössä:

```
LinkedList<String> ll = new LinkedList<String>();

// Adding elements to the linked list
ll.add("A");
ll.add("B");
ll.addLast("C");
ll.addFirst("D");
ll.add(2, "E");

ArrayList<String> al = new ArrayList<String>(ll);

System.out.println(al);
```

Ohjelma tulostaa:

```
[D, A, E, B, C]
```

Täten voidaan huomata, että konstruktori toimii myös, vaikka sille antaa LinkedList-olion. Tämä onnistuu, koska kyseiselle konstruktorille tulee antaa Collection-tyyppinen parametri. Koska LinkedListin määrittely toteuttaa Collection-rajapinnan, se voidaan tulkita myös Collection-olioksi tässä tapauksessa ja ohjelma voidaan suorittaa.

G.

Tämä ei toimi, sillä konstruktorille ei kelpaa parametriksi Array-tyyppinen olio:

```
// declares an Array of integers.
int[] arr;
// allocating memory for 5 integers.
arr = new int[5];
// initialize the first elements of the array
arr[0] = 10;
// initialize the second elements of the array
arr[1] = 20;
// so on...
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
ArrayList<Integer> al = new ArrayList<Integer>(arr); // saadaan virheilmoitus
```

Array-olio tulee muuttaa ArrayList-tyyppiseksi olioksi:

```
// declares an Array of integers.
int[] arr;
// allocating memory for 5 integers.
arr = new int[5];
// initialize the first elements of the array
arr[0] = 10;
// initialize the second elements of the array
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
List<Integer> al = new ArrayList<Integer>();
// adding elements of array to arrayList.
for (int i : arr) {
    al.add(i);
}
System.out.println(al);
```

Tässä tapauksessa ohjelma voidaan ajaa ja tulosteeksi saadaan:

```
[10, 20, 30, 40, 50]
```

H.

Jono voidaan toteuttaa esimerkiksi seuraavasti:

```
Queue<Integer> q = new LinkedList<>();
```

I.

Stack on luokka eikä rajapinta, kuten useimmat tietorakenteet. Koska Stack-luokka perii luokan Vector, eli se on Vector-luokan aliluokka, sillä on kyky käyttää Vector-luokan toimintoja. Stack luokka voi esimerkiksi käyttää, lisätä ja poistaa alkioita niiden indeksien perusteella Stack-luokalle tyypillisten metodien lisäksi.

J.

Suoritetaan seuraava ohjelma:

```
StringBuilder sb = new StringBuilder("Java");
String s = "Java";

s.concat(" 17");
sb.append(" 17");

System.out.println("String: " + s);
System.out.println("StringBuilder: " + sb);
```

Tulosteeksi saadaan:

```
String: Java
StringBuilder: Java 17
```

String-olio ei muutu, koska Javassa merkkijonotyyppiset oliot ovat mutatoimattomia, toisin kuin Stringbuilder-oliot. Jos haluamme kuitenkin käyttää kuitenkin concat()-metodia, tulee muuttuja "s" määrittää uudelleen:

```
s = s.concat(" 17");
```

Tehtävä 2

A.

Literaaliluokka eli enum-luokka. Luokan oliot ovat muuttumattomia, eikä niitä voi lisätä muuttamatta kyseisen luokan koodia. Tämän luokkakonstruktion avulla on helppo luoda ennalta määrätty määrä eri puulajeja. Jos käytettäisiin enum-luokan sijasta normaalia luokkaa, tulisi koodia kirjoittaa myös huomattavasti enemmän.

B.

Esiintymäkohtainen sisäluokka. Tämä luokkakonsturkito sopii parhaiten kyseiseen tarkoitukseen, sillä se vaatii ulkoluokasta luodun instanssin toimiakseen. Tässä tapauksessa esimerkiksi Ajoneuvo-olion, johon voidaan yhdistää sisäluokan määrittämän ajoneuvotyypin mukaiset ominaisuudet.

C.

Funktioliteralit ja rajapinnat. Tässä tapauksessa kyseisellä luokkakonstruktiolla on helppo toteuttaa metodi, jolle annetaan syötteeksi jokin arvo. Tässä tapauksessa pallon säde.

D.

Staattinen sisäluokka. Tämä luokkakonstruktio kokoaa tässä tapauksessa toisiinsa liittyvät ajoneuvoluokat yhden luokan alle. Tällöin ulkoisella luokalla, eli Kuormalaskin-luokalla on pääsy mainittuihin sisäluokkiin.

E.

Staattinen sisäluokka. Tätä luokkakonstruktiota käyttämällä on mahdollista luoda helposti avain-arvopareja pitäen samalla kirjaa niiden määrästä, jotta niiden ennalta määritettyä määrää ei ylitetä.

F.

Tietueluokka eli record-luokka. Tällä luokkakonstruktiolla voidaan mallintaa ei-polymorfista, eli muuttumatonta dataa. Tämä sopii luokkakonstruktioista tähän parhaiten, koska tietueiden ei haluta muuttuvan.

G.

Rajapintaluokka eli interface. Käyttämällä rajapintaa voimme määritellä kahden toiminnalisen osan vuorovaikutusta, ja samalla asettaa toiminnalliset luokat toteuttamaan rajapintaa. Täten varmasti molemmat luokat toteuttavat keskinäisen vuorovaikutuksen eli rajapinnan.

H

Nimetön luokka. Tämä sopii hyvin tarkoitukseen, koska kyseessä on yksinkertainen tapahtumankäsittelijä, jota tarvitaan kerran. Tällöin metodi voidaan määrittää tarvittaessa esimeriksi uudelleen. Lisäksi kyseinen luokka on kompakti kirjoittaa.

Tehtävä 4

A.

Hyvää luokkakaaviossa on se, että kaikki luokat ja rajapinnat ovat kuvattu ja niiden välisiä suhteita on selitetty. Kaavio on myös selkeä ja tarpeeksi yksinkertainen ymmärtää, vaikkakin tämä on enemmän seurausta kuvattavien komponenttien suhteeellisen pienestä määrästä. Jokainen nuoli on piirretty siten että ne eivät risteä toisten nuolien kanssa. Tämä parantaa luettavuutta huomattavasti. Myös kaikki suhteita kuvaavat nuolet on valittu oikein ja niiden suunta on oikea. Luokkien metodit ja muuttujat sekä niiden suojausmääreet vastaavat itse lähdekoodin toteutusta.

UML-mallinnuksessa on tapana piirtää suoria viivoja ja suoria kulmia, jos viivan täytyy mutkitella. Kyseisessä kaaviossa viivat ovat vinoja, ja vaikka tämä ei yksinkertaisessa kaaviossa haittaa, laajemmissa kaavioissa tämä voi heikentää luettavuutta. Hyvä tapa UML mallinnuksessa on sijoittaa yliluokat ylemmäs ja aliluokat näiden alle. Tätä ei ole kuitenkaan noudatettu, sillä luokka `TyökaluBase` on alimmaisena, ollessa kuitenkin yksittäistä työkalua edustavien luokkien yläluokka. Myöskään enum-luokkan `Metallit` suhdetta muihin luokkiin ei ole

kuvattu mitenkään, kuten myös luokka `Työkaluvarasto` on jäänyt ilman mitään yhteyksiä muihin luokkiin kaaviossa. Myöskään mitään lukumääräsuhteita luokkakaavio ei kuvaa, vaikka ne eivät ole välttämättä olennaisia tässä luokkakaaviossa.

Tehtävä 5

A.

`piirrä`-metodi määritellään julkisessa rajapintaluokassa `Piirros`. Koska piirrä metodi kuuluu rajapintaluokkaan, sillä ei ole toiminnallisuutta itsessään, vaan jokainen luokka joka rajapintaa käyttää, voi itse määritellä toteutuksen. `piirrä`-metodi on myös määritelty abstraktissa luokassa `Kuvio`. Abstraktissa luokassa määritelty `piirä`-metodi on myös määrittelemätön ja se voidaan määritellä perijäluokan toimesta `@Override`-avainsanalla. Täten tässä tapauksessa ilmenee **polymorfismia**. Tällöin asiakas voi käsitellä perijäluokan metodia edes havaitsematta sitä.

B.

Molemmat toteutukset `Piirros1` ja `Piirros2` toteuttavat rajapintaluokan `Piirros`, mutta luokka `Piirros2` perii myös luokan `Siksak`. `Siksak`-luokka toteuttaa rajapinnan `Asettelu`. `Piirros1`-luokassa luodaan uusi instanssi `Siksak`-oliosta ilman että `Piirros1`-luokka varsinaisesti perii kyseistä luokkaa.

Luokan `Piiros2` kanssa on kyse **periytymisestä**, koska kyseinen luokka perii kaikki `Siksak`-luokan ominaisuudet. `Piirros1`-luokan tapauksessa on kyse vain uuden `Siksak`-olion luomisesta, eli **kompositiosta**, kun `rakenna`-metodissa luodaan uudet `Siksak`- ja `Neliöt`-instanssit. `Piirros1`-luokassa myös `piirrä`-metodi toimiii eritavalla, sillä esimeriksi piste kutsutaan eritavalla.

Etuna `Piirros1`-luokassa on se, että sen tapauksessa rajapintasitoumukset ovat helpommin valittavissa. Kompositio on kuitenkin usein pidempi kirjoittaa perityminen ja sitä käytettäessä tulee olla täysi ymmärrys luotujen olioiden toimminnasta. `Piirros2`:en tapauksessa perimällä `Siksak`-luokka, voidaan asettelu ulkoistaa toiselle luokalle ja käyttää jo kirjoitettua koodia uudelleen. Periytymisen tapauksessa metodeja voidaan myös korvata, kuten luokan `Piirros2` tapauksessa.

C.

Luokilla `Piirros2` ja `Piirros3` on molemmilla sama singatuuri lukuunottamatta itse niiden nimeä, mutta niiden toiminnallisuudet ovat erilaiset. Molempien `piirrä`-metodit ovat kuitenkin samanlaiset.

Piirros3 on määritelty `extends Siksak implements Piirros` eikä `extends Siksak, Neliö`, koska `Piirros3`-luokka hyödyntää myös **kompositiota** rivillä 70 `Kuviointi kuviointi = new Neliöt()`, jossa luodaan uusi `Kuviointi`-olio, uuden `Neliöt`-olion avulla, vaikka `Piirros3`-luokka ei peri `Neliöt`-luokkaa. Tämä on mahdollista, koska `Neliöt`-luokka toteuttaa rajapinnan `Kuviointi`. Täten `Neliöt` oliota voidaan kuvata myös `Kuviot` oliona. Luokka `Piirros3` hyödyntää siis **Dynaamista sidontaa**.

Kun dynaamisen sidonta yhdistetään polymorfismin ja rajapintamäärittelyn kanssa, voidaan olioita käsitellä abstraktimmin eri konteksteissa tietämättä kuitenkaan niiden konkreettisia piirteitä.

`Piirros2`-luokka on lyhyempi kirjoittaa, kuin luokka `Piirros3`.