

Note this worksheet is very long and is not expected to be finished in an hour.

1 Athletes

Suppose we have the Person, Athlete, and SoccerPlayer classes defined below.

```
1 class Person {  
2     void speakTo(Person other) { System.out.println("kudos"); }  
3     void watch(SoccerPlayer other) { System.out.println("wow"); }  
4 }  
5  
6 class Athlete extends Person {  
7     void speakTo(Athlete other) { System.out.println("take notes"); }  
8     void watch(Athlete other) { System.out.println("game on"); }  
9 }  
10  
11 class SoccerPlayer extends Athlete {  
12     void speakTo(Athlete other) { System.out.println("respect"); }  
13     void speakTo(Person other) { System.out.println("hmpf"); }  
14 }
```

- (a) For each line below, write what, if anything, is printed after its execution.
Write CE if there is a compiler error and RE if there is a runtime error. If a line errors, continue executing the rest of the lines.

```
1 Person itai = new Person();  
2  
3 SoccerPlayer shivani = new Person();  
4  
5 Athlete sohum = new SoccerPlayer();  
6  
7 Person jack = new Athlete();  
8  
9 Athlete anjali = new Athlete();  
10  
11 SoccerPlayer chirasree = new SoccerPlayer();  
12  
13 itai.watch(chirasree);  
14  
15 jack.watch(sohum);  
16  
17 itai.speakTo(sohum);  
18
```

```

19 jack.speakTo(anjali);
20
21 anjali.speakTo(chirasree);
22
23 sohum.speakTo(itai);
24
25 chirasree.speakTo((SoccerPlayer) sohum);
26
27 sohum.watch(itai);
28
29 sohum.watch((Athlete) itai);
30
31 ((Athlete) jack).speakTo(anjali);
32
33 ((SoccerPlayer) jack).speakTo(chirasree);
34
35 ((Person) chirasree).speakTo(itai);

```

Solution: Here is a video walkthrough of the solution.

```

Person itai = new Person();
SoccerPlayer shivani = new Person(); // CE
Athlete sohum = new SoccerPlayer();
Person jack = new Athlete();
Athlete anjali = new Athlete();
SoccerPlayer chirasree = new SoccerPlayer();
itai.watch(chirasree); // wow
jack.watch(sohum); // CE
itai.speakTo(sohum); // kudos
jack.speakTo(anjali); // kudos
anjali.speakTo(chirasree); // take notes
sohum.speakTo(itai); // hmpf
chirasree.speakTo((SoccerPlayer) sohum); // respect
sohum.watch(itai); // CE
sohum.watch((Athlete) itai); // RE
((Athlete) jack).speakTo(anjali); // take notes
((SoccerPlayer) jack).speakTo(chirasree); // RE
((Person) chirasree).speakTo(itai); // hmpf

```

Explanation:

Line 3: Person is a superclass of SoccerPlayer, so it can't be assigned to a variable of type SoccerPlayer. (In general, an object can be assigned to a variable that is the same class or a superclass of it).

Line 13: itai has the same static and dynamic type (Person) and Person.watch is allowed to take in a SoccerPlayer argument, so we use that method and print wow.

Line 15: jack has static type Person and dynamic type Athlete. sohum has static type Athlete (we only care about the static type of arguments). During

compile time, we choose `Person.watch`, which can only take in a `SoccerPlayer`. `Athlete` is a superclass of `SoccerPlayer`, so this method cannot take in an `Athlete` and a compilation error results.

Line 17: `sohum` has static type `Athlete`, and `itai` has static and dynamic type `Person`, so we must use `Person.speakTo`. `Person.speakTo` takes in a `Person`, a superclass of `sohum`'s type, so this method works.

Line 19: `jack` has static type `Person`, dynamic type `Athlete`. `anjali` has static type `Athlete`. During compile time, we choose the method signature `speakTo(Person other)`. During runtime, we check the class of `jack`'s dynamic type. `Athlete` does not have a method matching our earlier signature, so we use our earlier method and print `kudos`.

Line 21: `anjali` has static and dynamic type `Athlete`. `chirasree` has static type `SoccerPlayer`. The only method we can use is `Athlete.speakTo`. This is fine because `SoccerPlayer` is a subclass of `Athlete`, so we print `take notes`.

Line 23: `sohum` has static type `Athlete` and dynamic type `SoccerPlayer`. `itai` has static type `Person`. During compilation, we first go to `Athlete`. However, `Athlete.speakTo` cannot take in a `Person`, so we go to its parent, `Person`, and choose the signature `speakTo(Person other)`. Then, during runtime, we check `sohum`'s dynamic type, `SoccerPlayer`. `SoccerPlayer.speakTo(Person other)` matches our signature, so we use that method and print `hmpf`.

Line 25: `chirasree` has static and dynamic type `SoccerPlayer`. We call the `SoccerPlayer.speakTo` method with an argument of type `SoccerPlayer`, which selects the most specific signature possible—`SoccerPlayer.speakTo(Athlete other)`, printing `respect`.

Line 27: `sohum` has static type `Athlete` and dynamic type `SoccerPlayer`. `itai` has static type `Person`. During compile time, we go to `Athlete`, but `Athlete.watch(Athlete other)` cannot handle an argument of type `Person`, so we go to its parent. However, `Person.watch(SoccerPlayer other)` also cannot handle an argument of type `Person`, so this results in a compilation error.

Line 29: The compiler "trusts" that the cast of `itai` to `Athlete` is correct; however, during runtime, casting a `Person` to an `Athlete` fails, resulting in a runtime exception.

Line 31: By casting, we tell the compiler to view `jack`'s static type as `Athlete`. Thus, during compilation, we choose the signature `Athlete.speakTo(Athlete other)`. Then, during runtime, `jack` has dynamic type `Athlete`, so the cast is valid, and we print `take notes`.

Line 33: `Jack` has dynamic type `Athlete`, which cannot be downcast to a subclass `SoccerPlayer`.

Line 35: During compilation, we treat `chirasree` as a `Person` and choose the method signature `speakTo(Person other)`. Then, during runtime, we see that `chirasree` has dynamic type `SoccerPlayer`, so we choose the `SoccerPlayer.speakTo(Person other)` method that matches our earlier signature, and print `hmpf`.

- (b) You may have noticed that `jack.watch(sohum)` produces a compile error. Interestingly, we can resolve this error by **adding casting!** List two fixes that would resolve this error. The first fix should print `wow`. The second fix should

print game on. Each fix may cast either jack or sohum.

- 1.
- 2.

Solution: Here is a video walkthrough of the solutions for this part and the next.

1. To print wow, we can cast sohum as a SoccerPlayer, resulting in the function call jack.watch((SoccerPlayer) sohum);
2. To print game on, we can cast jack as an Athlete, resulting in the function call ((Athlete) jack).watch(sohum);

- (c) Now let's try resolving as many of the remaining errors from above by **adding or removing casting!** For each error that can be resolved with casting, write the modified function call below. Note that you cannot resolve a compile error by creating a runtime error! Also note that not all, or any, of the errors may be resolved.

Solution:

```
jack.speakTo(chirasree);
```

Explanation: This resolves the casting error on line 33. jack has static type person, and Person.speakTo(Person other) can handle an argument of type SoccerPlayer, so no compilation errors are produced either.

2 Hidden Fruits

Suppose we have the `Fruit` and `Persimmon` and classes defined below.

```

1  class Fruit {
2      String flavor = "generic";
3      static char start = 'f';
4
5      static int eat(Fruit fruit) {
6          return 1;
7      }
8
9      char hats() {
10         return this.start;
11     }
12 }
13
14 class Persimmon extends Fruit {
15     String flavor = "superb";
16     static char start = 'p';
17
18     static int eat(Fruit fruit) {
19         return 2;
20     }
21
22     int eat(Persimmon persimmon) {
23         return 3;
24     }
25 }
```

For each line below, write what, if anything, is printed after its execution. Write CE if there is a compiler error and RE if there is a runtime error. If a line errors, continue executing the rest of the lines.

```

1 Fruit shreyas = new Fruit();
2 Fruit aram = new Persimmon();
3 Persimmon eric = new Persimmon();
4
5 System.out.println(eric.flavor);
6 System.out.println(aram.flavor);
7
8 System.out.println(eric.eat(shreyas));
9 System.out.println(eric.eat(eric));
10 System.out.println(aram.eat(eric));
11
12 System.out.println(aram.hats());
13 System.out.println(eric.hats());
```

Solution:

```
Fruit shreyas = new Fruit();
Fruit aram = new Persimmon();
Persimmon eric = new Persimmon();

System.out.println(eric.flavor); // superb
System.out.println(aram.flavor); // generic

System.out.println(eric.eat(shreyas)); // 2
System.out.println(eric.eat(eric)); // 3
System.out.println(aram.eat(eric)); // 1

System.out.println(aram.hats()); // f
System.out.println(eric.hats()); // f
```

3 Containers

a) (1 Points). Suppose that we have the Container abstract class below, with the abstract method `pour` and the method `drain`. Implement the method `drain` so that all the liquid is drained from the container, i.e. `amountFilled` is set to 0. Return `true` if any liquid was drained, and `false` otherwise. In other words, return `true` if and only if there is liquid in the container prior to the function being called. You may add a maximum of **5 lines of code**. Note that the staff solution uses 3. You may *only* add code to the `drain` method. (Summer 2021 MT1)

```

1  public abstract class Container {
2      /* Keeps track of the total amount of liquid in the container */
3      public int amountFilled;
4
5      public boolean drain() {
6
7
8
9
10
11 } // You may use at most 5 lines of code, i.e. this bracket should be on line 11 or earlier.
12
13     abstract int pour(int amount);
14 }
```

Solution: Here is a video walkthrough of the solution.

```

1  public abstract class Container {
2      /* Keeps track of the total amount of liquid in the container */
3      public int amountFilled;
4
5      public boolean drain() {
6          boolean answer = amountFilled > 0;
7          amountFilled = 0;
8          return answer;
9      }
10
11     abstract int pour(int amount);
12 }
```

b) (1.5 Points). Finish implementing the `WaterBottle` class so that it **is a** Container. You should *only* add code to the blanks, i.e. **fill in the `pour` method and the class signature**.

As stated in the `Container` class, the `pour` method should pour `amount` into the container and return the amount of the excess liquid, or 0 if there is no excess. For instance, suppose we have a `WaterBottle w` with `capacity 10` and `amountFilled 5`. Then, if we execute `w.pour(7)`, `amountFilled` should be set to **10** and **2** should be returned. Your solution *must* fit within the blanks provided. You may not need all

the lines.

```

1 class WaterBottle extends Container {
2     private static final int DEFAULT_CAPACITY = 16;
3
4     /* The capacity of the container, i.e. the maximum amount of liquid the water bottle can hold */
5     private int capacity;
6
7     WaterBottle() {
8         this(DEFAULT_CAPACITY);
9     }
10    WaterBottle(int capacity) {
11        this.capacity = capacity;
12        this.amountFilled = 0;
13    }
14
15    @Override
16    public int pour(int amount) {
17        _____;
18        if (_____) {
19            _____;
20            _____;
21            _____;
22        }
23        _____;
24    }
25 }
```

Solution: Here is a video walkthrough of the solution.

```

1 class WaterBottle extends Container {
2     private static final int DEFAULT_CAPACITY = 16;
3
4     /* The capacity of the container, i.e. the maximum amount of liquid the water bottle can hold */
5     private int capacity;
6
7     WaterBottle() {
8         this(DEFAULT_CAPACITY);
9     }
10    WaterBottle(int capacity) {
11        this.capacity = capacity;
12        this.amountFilled = 0;
13    }
14
15    @Override
16    public int pour(int amount) {
17        filled += amount;
18        if (filled > capacity) {
```

```

19         int excesss = filled - capacity;
20         filled = capacity;
21         return excesss;
22     }
23     return 0;
24 }
25 }
```

c) (4 Points). Finally, suppose we have the `ContainerList` class, with the `drainFirst` method as implemented below. Unfortunately, the `drainFirst` method *sometimes* errors!

In order to fix it, you may add code to the **ContainerList constructor** and the **UnknownContainer** class! You may only **use** 5 lines of code in the `ContainerList` constructor and **add** 4 lines of code to the `UnknownContainer` class! If you decide to keep or modify the given line in the `ContainerList` constructor, it counts as one of the 5 lines.

Note that, after making your changes, the `drainFirst` should **never error** and **retain the functionality in the docstring**. You **may not modify the `drainFirst` method!** You may use classes from the previous part assuming they are implemented correctly.

Hint: Make sure that, with your fix, the `drainFirst` method won't error, even if the `drainFirst` method is called many times.

```

1 class UnknownContainer {
2     // TODO
3
4
5
6
7
8 } // You may add at most 4 lines of code to the class above
9 // i.e. the closing bracket should be on line 6 or earlier
10
11 class ContainerList {
12     private Container[] containers;
13
14     ContainerList(Container[] conts) {
15         this.containers = conts; // you may delete, modify, or keep this line
16         // YOUR CODE HERE
17
18
19
20
21
22 } // You may use at most 5 lines of code in the Constructor
```

```

23     // i.e. the closing bracket should be on line 18 or earlier
24
25     /* Drains the water from the first nonempty container */
26     void drainFirst() {
27         int index = 0;
28         while (!containers[index].drain()) {
29             index += 1;
30         }
31     }
32 }
```

Solution: Here is a video walkthrough of the solution.

```

1  class UnknownContainer extends WaterBottle {
2      @Override
3      public boolean drain() {
4          return true;
5      }
6  }
7
8  class ContainerList {
9      private Container[] containers;
10
11     ContainerList(Container[] conts) {
12         containers = new Container[conts.length + 1];
13         for (int i = 0; i < conts.length; i += 1) {
14             containers[i] = conts[i];
15         }
16         //     System.arraycopy(conts, 0, containers, 0, conts.length); <- can replace for loop with this
17         containers[conts.length] = new UnknownContainer();
18     }
19
20     /* Drains the first nonempty container */
21     void drainFirst() {
22         int index = 0;
23         while (!containers[index].drain()) {
24             index += 1;
25         }
26     }
27 }
```

Explanation: `drainFirst` cannot handle a case with all empty containers—it keeps incrementing `index` until it's out of bounds. The solution is to add a `Container` which can always be drained, the `UnknownContainer`. Thus, we write an `UnknownContainer.drain` which always returns `true`.

However, we can't just override `Container`, since this will require you to implement both `drain` and `pour` (which requires more than 4 lines). Instead, we have to extend `WaterBottle`.

Then, in `ContainerList`, we add an extra `UnknownContainer` to the end of the `containers` list. However, an array's size cannot be changed, so we have to copy `conts`, then add the last `UnknownContainer`.

The following two problems are very challenging, and we only recommend attempting after finishing the rest of the worksheet.

4 Challenge: Frauds List

(6 Points). Suppose we have the `IntList` and `FraudList` classes below (Summer 2021, Final)

```

1  public class IntList {
2      public int first;
3      public IntList rest;
4
5      public IntList(int f, IntList r) {
6          first = f;
7          rest = r;
8      }
9
10     public int size() {
11         IntList p = this;
12         int totalSize = 0;
13         while (p != null) {
14             totalSize += 1;
15             p = p.rest;
16         }
17         return totalSize;
18     }
19 }
20
21 class FraudList extends IntList {
22     public FraudList(int f, IntList r) {
23         super(f, r);
24     }
25     public int size() {
26         return -super.size();
27     }
28 }
```

Implement the method `findFrauds` which accepts an array of `IntLists` in which some of the elements are, or may contain, `FraudLists`! That is, the dynamic type of certain `IntList` instances is `FraudList`. As shown above, a `FraudList` is an `IntList` whose size method returns the negative of the correct size. You must report these `FraudLists` by **non-destructively** returning a `new FraudList` of all the `FraudList` instances linked together in the order they appear in `arr`.

You may **not** modify the given array `arr` or the `IntLists` inside of `FraudList`. You may **not** use `instanceOf`, `getClass()`, `isInstance()` or any method not explicitly written in the classes above or imported. An instance of the problem is shown below:

```

1 IntList first = new IntList(1000, new IntList(1002, new FraudList(1, new FraudList(2, null))));
2 IntList second = new FraudList(3, null);
3 IntList third = new IntList(3000, null);
4 IntList fourth = new FraudList(4, new IntList(231, new FraudList(5, null)));
5 IntList[] arr = new IntList[]{first, second, third, fourth};
6 FraudList frauds = findFrauds(arr);

```

After executing the lines above, `frauds` should be equal to the `FraudList` with the elements 1, 2, 3, 4, 5 and `arr`, as well as the contents within `arr`, should be unchanged. Fill in the skeleton below. You may not delete, modify, or add to any of the provided skeleton code.

```

1 import static java.lang.System.arraycopy;
2
3 public static FraudList findFrauds(IntList[] arr) {
4     IntList[] copy = new IntList[arr.length];
5     arraycopy(arr, 0, copy, 0, arr.length);
6     return helper(copy, 0);
7 }
8
9 public static FraudList helper(IntList[] copy, int index) {
10    if (index == copy.length) {
11        return null;
12    } else if (copy[index] instanceof FraudList) {
13        return (FraudList) copy[index];
14    }
15    copy[index] = helper(copy, index + 1);
16    copy[index] = helper(copy, index + 1);
17    if (copy[index] instanceof FraudList) {
18        return copy[index];
19    } else {
20        return copy[index];
21    }
22 }

```

Solution:

```

1 import static java.lang.System.arraycopy;
2
3 public static FraudList findFrauds(IntList[] arr) {
4     IntList[] copy = new IntList[arr.length];
5     arraycopy(arr, 0, copy, 0, arr.length);
6     return helper(copy, 0);
7 }
8
9 public static FraudList helper(IntList[] copy, int index) {
10    if (index == copy.length) {
11        return null;
12    }
13    if (copy[index] instanceof FraudList) {
14        return copy[index];
15    } else {
16        return copy[index];
17    }
18 }

```

```
12     } else if (copy[index] == null) {
13         return helper(copy, index + 1);
14     }
15     IntList current = copy[index];
16     copy[index] = current.rest;
17     if (current.size() < 0) {
18         return new FraudList(current.first, helper(copy, index));
19     } else {
20         return helper(copy, index);
21     }
22 }
```

5 Challenge: A Puzzle

Consider the partially filled classes for A and B as defined below:

```

1  public class A {
2      public static void main(String[] args) {
3          ___ y = new ___();
4          ___ z = new ___();
5      }
6
7      int fish(A other) {
8          return 1;
9      }
10
11     int fish(B other) {
12         return 2;
13     }
14 }
15
16 class B extends A {
17     @Override
18     int fish(B other) {
19         return 3;
20     }
21 }
```

Note that the only missing pieces of the classes above are static/dynamic types! Fill in the **four** blanks with the appropriate static/dynamic type — A or B — such that the following are true:

1. y.fish(z) equals z.fish(z)
2. z.fish(y) equals y.fish(y)
3. z.fish(z) does not equal y.fish(y)

Solution: Here is a video walkthrough of the solutions.

```

1  public class A {
2      public static void main(String[] args) {
3          A y = new B();
4          B z = new B();
5      }
6      ...
7  }
```

Explanation: To get to this solution, it's helpful to write a matrix of possible static/dynamic types, and eliminate ones that don't work. First note that because of (3), y and z cannot both be static type B; otherwise only B.fish(B other) would ever get called. Also, they cannot both have static type A: method arguments only check static types, so only A.fish(A other) would ever get called, violating (3). Since we know A and B must have different static types, let's try assigning static

type A to y and static type B to z. (z must also have dynamic type B, since an object's dynamic type either the same as or a subclass of it's static type). Checking the result of `y.fish(z)`, we see that this will choose the method signature `fish(B other)` inside A at compile time. However, for `z.fish(z)`, the compiler goes to B and chooses `B.fish(B other)`. In order for these two method calls to be equal, the dynamic type of y must be B.

This gives us our final answer: y has static type A, dynamic type B; and z has static and dynamic type B. We check (2) to make sure this works. `z.fish(y)` will go to B first, but since B only has a method for `fish(B other)`, we must go to it's superclass and choose `fish(A other)` in A at compile time. `y.fish(y)` choose the same method, `A.fish(A other)`. During runtime, we check the dynamic type of z, B, which does not have a matching signature, so both these calls return 2 as desired.