Part 1 (10 Points): True/False Questions

1. (T) Immutable Objects are automatically thread safe. So you don't need any synchronization.
2. (F) You can declare a generic method in a non generic class but you cannot declare a generic interface without declaring a generic method in it.
3. (F) Since Object is a supertype of all reference types in Java, so we can write code like this:
   ```
   Unit<String> us = new Unit<>();
   Unit<Object> uo = us;
   ```
4. (F) Both Adder and SmartAdder methods below are functional interface because both of them contain only one abstract method and the abstract method is not the method defined in Object class.
   ```
   public interface Adder {
           int add (int a, int b);
   }
   public interface SmartAdder extends Adder {
           int add (double a, double b);
   }
   ```
5. (F) The following code will compile without problems since Java only support single inheritance for class and Object is the super-type of all classes even though we omit "extends Object" most of the time.
   ```
   public enun Size extends Object {
           SMALL, MEDIUM, LARGE }
   ```
6. (F) In Java 8, Stream's limit() is a terminal operation because it will return a stream of a limited size.
7. (F) If two objects have the same hashcode, the equals() must return true.
8. (F) This is a correct lambda expression: ()-> return "Tricky Exam".
9. (T) The following statements can compile without errors. CheckingAccount is a subclass of Account.
   ```
   List<? extends CheckingAccount> checkingAccounts= new ArrayList<>();
   List<? extends Account> accounts = checkingAccounts;
   ```
10. (F) The following lambda expression is correct.
    ```
    Object o = () -> { System.out.println("Tricky Final Exam!"); };
    ```

PART II (20 points): Short Answer Questions, please write your answer clearly.

11. (**4 Points**) Name two differences between List<?> and List<Object>.

    (**4 Points**) What are the differences between List<?> and List<Object> in Java (at least two)?

    a. List<?> is a list of unknown type. List<Object> is Object type list.
    b. List<?> is a supertype of all generic list, but List<Object> is not.
    c. List<Object> can hold any type of object, whereas a List<?> can only hold objects of a specific type or subtype.

12. (**4 Points**) How to make a class immutable?

    a. Make all attributes private & final.
    b. Make the class final.
    c. No setters, only allow getters.
    d. Don't send mutable objects through getters. Clone () method to use in that case.

13. (**4 Points**) When we use instance-of-strategy to override equals(). If a subclass is introduced subclass inherits the equals method. If subclass overrides equals, then an asymmetric equals is created. In this case, how do you solve the problem? (At least two ways)

    a. Use the getClass() method to compare the classes of the two objects before checking their fields for equality. This ensures that the equals() method will return false if the two objects being compared are not of the same class.
    b. Use the instanceof operator to check if the other object is of the same type as this object before comparing the fields. This will avoid the possibility of ClassCastExceptions.
    c. Use the Objects.equals() method to compare the fields of the two objects for equality. This method takes care of null values and handles the comparison of arrays as well.
    d. Use the final keyword in the declaration of the equals() method to prevent subclasses from overriding it.
    e. Override the hashCode() method along with the equals() method, and ensure that the same fields are used for both methods. This will ensure that two objects that are equal according to the equals() method will have the same hash code

14. (**4 Points**) List at least 4 features / characteristics of functional style of programming.

a. Immutability: Functional programming emphasizes the use of immutable data structures and objects. This means that once a value is assigned, it cannot be changed.
b. Functions as first-class citizens: Functions in functional programming are treated as first-class citizens, which means they can be assigned to variables, passed as arguments to other functions, and returned as values from functions.
c. Pure functions: A pure function is a function that has no side effects and always returns the same output for the same input. It does not modify any data outside of its scope.
d. Higher-order functions: Higher-order functions are functions that take other functions as arguments or return a function as a result.
e. Lazy evaluation: In functional programming, lazy evaluation is used to evaluate expressions only when their results are needed, rather than computing everything all at once.
f. Recursion: Recursion is heavily used in functional programming, where loops are replaced with recursive functions.
g. Declarative programming: Functional programming is often referred to as declarative programming, where the focus is on "what" needs to be done, rather than "how" to do it.
h. Implied control flow: In functional programming, the control flow is implied by the data flow, rather than being explicitly defined by control structures like loops and conditionals.
i. Composition: Functional programming emphasizes the composition of small, reusable functions to build larger and more complex programs.
j. Concurrency: Functional programming encourages the use of concurrency, where multiple tasks can be executed simultaneously and independently, without causing interference with each other.

15. (**4 Points**) When you assign appropriate type, you have to specify type argument which means change type variable T to specific type such as Integer, Employee, etc.

    a. (**2 Points**) Assign the appropriate type to the method reference below and convert it to lambda expression.

    String::compareTo

    Ans: Method Reference: BiFunction<String, String, Integer>

    Lambda expression: (s1,s2) -> s1. compareTo (s2)

    b. Assign the appropriate type to the method reference below and convert it to method reference.

    () -> new HashSet<Person>();

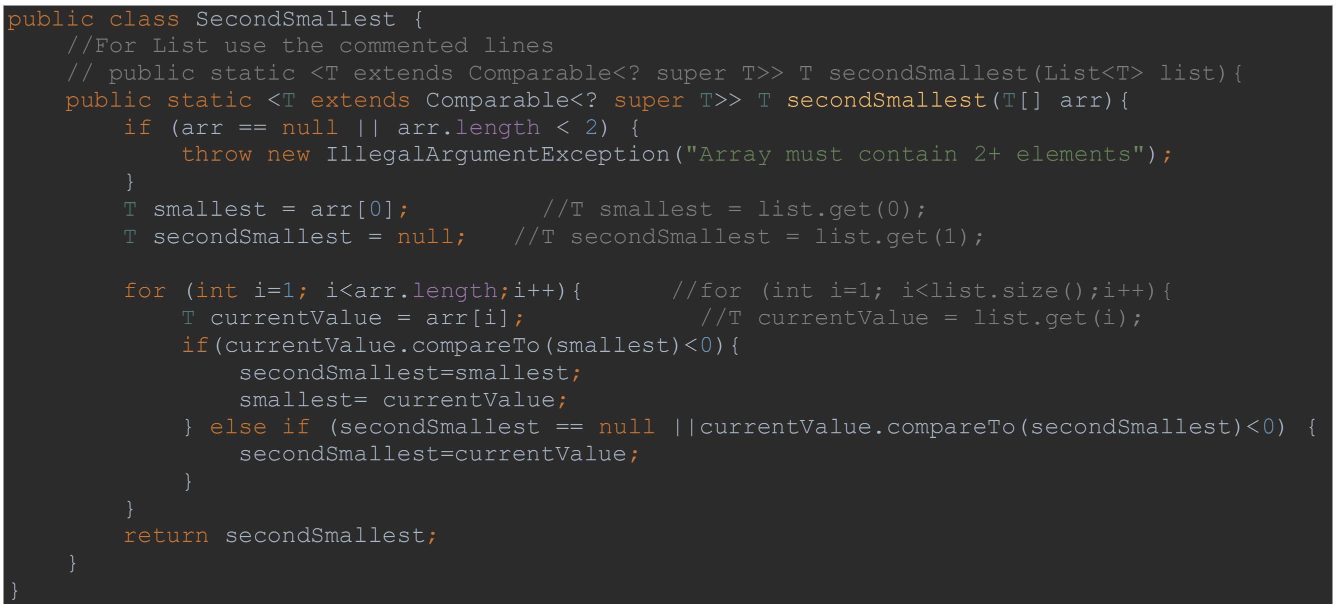
    Ans: Method Reference: HashSet<Person>::new

    Type: Supplier<HashSet<Person>>

    Lambda Expression: () -> new HashSet<Person>()

PART II (60 points): Programming Questions

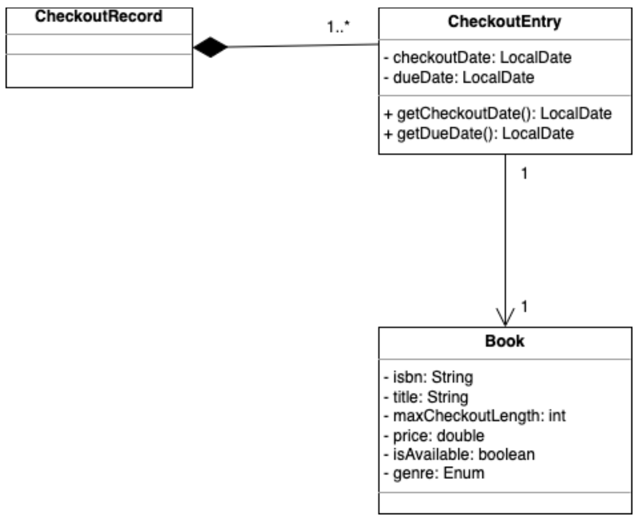
16. (**10 Points**) Create a generic programming solution to the problem of finding the second smallest element in an Array. In other words, devise a public static method secondSmallest so that it can handle the biggest possible range of types (such as Integer, Double, Number, LocalDate, etc). You only need to write the generic method.
    Test cases:
    a. 1,2,3,4,5 result 2
    b. 1,1,1,2,2,3,4,4,5 result 2
    c. LocalDate: 1900-1-1, 1950-1-1, 1975-1-1, 2000-1-1 result 1950-1-1



```java
public class SecondSmallest {
    //For List use the commented lines
    // public static <T extends Comparable<? super T>> T secondSmallest(List<T> list){
    public static <T extends Comparable<? super T>> T secondSmallest(T[] arr){
        if (arr == null || arr.length < 2) {
            throw new IllegalArgumentException("Array must contain 2+ elements");
        }
        T smallest = arr[0];          //T smallest = list.get(0);
        T secondSmallest = null;    //T secondSmallest = list.get(1);

        for (int i=1; i<arr.length;i++){       //for (int i=1; i<list.size();i++){
            T currentValue = arr[i];             //T currentValue = list.get(i);
            if(currentValue.compareTo(smallest)<0){
                secondSmallest=smallest;
                smallest= currentValue;
            } else if (secondSmallest == null ||currentValue.compareTo(secondSmallest)<0) {
                secondSmallest=currentValue;
            }
        }
        return secondSmallest;
    }
}
```

17. (**25 Points**) Use the class diagram and the code provided to help you solve the problems given below.



```
public class CheckoutRecord {
  private List<CheckoutRecordEntry> checkoutEntries = new ArrayList<>();
  public List<CheckoutRecordEntry> getCheckoutEntryList() {
    return checkoutEntries;
  }
  public CheckoutRecord() {
  }
  public void addEntry(CheckoutRecordEntry entry) {
    checkoutEntries.add(entry);
  }
  @Override
  public String toString() {
    return checkoutEntries. toString();
  }
}
```

```
public enum Genre {
  COMEDY, DRAMA, SCIFICTION, ROMANCE, TRAGEDY, SATIRE, IT;
}
```

```
public class CheckoutRecordEntry {
   private Book book;
   private LocalDate checkoutDate;
   private LocalDate dueDate;

   public CheckoutRecordEntry(Book copy, LocalDate checkoutDate, LocalDate dueDate) {
      this.book=copy;
      this.checkoutDate=checkoutDate;
      this.dueDate=dueDate;
```

```
    }
    //Omit setters and getters
    @Override
    public String toString() {
        String pattern = "MM/dd/yyyy";
        return "[" + book + " checkout date: "
                + checkoutDate.format(DateTimeFormatter.ofPattern(pattern))
                + " due date: "
                + dueDate.format(DateTimeFormatter.ofPattern(pattern)) + "]";
    }
}
```

```
public class Book {
    private String isbn;
    private String title;
    private int maxCheckoutLength;
    private double price;
    private boolean isAvailable;
    private Genre genre;

    public Book(String isbn, String title, int maxCheckoutLength,
                double price, boolean isAvailable, Genre genre) {
        this.isbn = isbn;
        this.title = title;
        this.maxCheckoutLength = maxCheckoutLength;
        this.price = price;
        this.isAvailable = isAvailable;
        this.genre = genre;
    }
    //Omit setters and getters
    @Override
    public String tostring() {
        return "Book [isbn=" + isbn + ", title=" + title
                + ", maxCheckoutlength=" + maxCheckoutLength + "price=" + price
                + ", isAvailable=" + isAvailable + "genre=" + genre + "]";
    }
}
```

```
public class Test {
   public static void main(String[] args) {
      List<Book> allBooks = Arrays.asList(
            new Book("1040A5", "Gone with the Wind", 5, 100.0, true, Genre.IT),
            new Book("5241C3", "The Unhappy Indian", 5, 100.0, true, Genre.IT),
            new Book("9982D1", "The Unseen Rock", 5, 100.0, true, Genre.IT),
            new Book("1112E5", "Hunting", 5, 100.0, true, Genre.IT),
            new Book("8008T4", "Overdrive", 5, 100.0, true, Genre.IT)
        );
        List<CheckoutRecordEntry> allEntries = Arrays.asList(
            new CheckoutRecordEntry(allBooks.get(0), LocalDate.of(2017,1,1), LocalDate.of(2017,3,1)),
            new CheckoutRecordEntry(allBooks.get(1), LocalDate.of(2016,1,6), LocalDate.of(2016,3,15)),
            new CheckoutRecordEntry(allBooks.get(2), LocalDate.of(2017,2,1), LocalDate.of(2017,4,1)),
            new CheckoutRecordEntry(allBooks.get(3), LocalDate.of(2017,4,1), LocalDate.of(2017,6,1)),
        );
    }
}
```

1) (**5 Points**) Given the lambda expression below:

```
public class checkoutRecord {
        //The lambda expression: (instance) -> this.equals(instance);
        //The lambda expression exists in CheckoutRecord class.
}
```

Provide an anonymous inner class that behaves the same way as the lambda.

Hint: find the functional interface first.

```
public class checkoutRecord {
    Predicate<checkoutRecord> p = new Predicate<checkoutRecord>() {
        @Override
        public boolean test(checkoutRecord instance) {
            return checkoutRecord.this.equals(instance);
        }
    };
}
```

2) (**5 Points**) Create a stream pipeline that, when run, finds all CheckoutRecordEntry for which dueDate is 2016-2-3 and sort them by checkoutDate.

```
List<CheckoutRecordEntry> result = allentries.stream()
          .filter(entry -> entry.getDueDate()
          .equals(LocalDate.of(2016, 2, 3)))
          .sorted(Comparator.comparing(CheckoutRecordEntry::getCheckoutDate))
          .collect(Collectors.toList());
```

3) (**5 points**) Create a stream pipeline that, when run ***returns a String*** of all books titles whose genre is ROMANCE, sorted alphabetically. The result's format books like: Allen, Seay's First Day of School, Siny's First Day of School,:

```
String result = books.stream()
         .filter(book -> book.getGenre() == Genre.ROMANCE)
         .sorted(Comparator.comparing(Book::getTitle))
         .map(Book::getTitle)
         .collect(Collectors.joining(", "));
```

4) (5 points) Create a stream pipeline that, when run, ***finds a book*** in CheckoatRecordEntry list with the cheapest price. (You **MUST** use reduce method)

```
Optional<Book> cheapestBook = checkoutRecordEntryList.stream()
        .flatMap(entry -> entry.getBooks().stream()) // convert to stream of books
        .reduce((book1, book2) -> book1.getPrice() < book2.getPrice() ? book1 :
book2); // find the cheapest book
```

5) (**5 points**) Turn the stream pipeline of Question 3) into a Library element for a Lambda Library which can be used to search based on other CheckoutRecordEntry lists and different Genre filter criteria. Make sure you choose the correct functional interface.

```
BiFunction<List<CheckoutRecordEntry>, Genre, String> searchBookTitlesByGenre =
        (checkoutRecordEntries, genre) -> checkoutRecordEntries.stream()
                        .filter(entry -> entry.getBook().getGenre()==genre)
                        .map(entry -> entry.getBook().getTitle())
                        .sorted().collect(Collectors.joining(", "));
```