*Generics and collections*

REVIEW NOTES

This section lists the main points covered in this chapter.

*Creating generic entities*

* You define a generic class, interface, or method by adding one or more type parameters to it.
* A class that uses a generic class uses a parameterized type, replacing the formal parameter with an actual parameter. Also, invalid casts aren’t allowed.
* Java’s naming conventions limit the use of single uppercase letters for type parameters. Though not recommended, using any valid identifier name for type parameters is acceptable code.
* A generic class can be extended by another generic or non-generic class.
* An extended class must be able to pass type arguments to its generic base class. If it doesn’t, the code won’t compile.
* When a non-generic class extends a generic class, the derived class doesn’t define any type parameters but passes arguments to all type parameters of its generic base class.
* A generic interface is defined by including one or more type parameters in its declaration.
* When a non-generic class implements a generic interface, the type parameters follow the interface name.
* When a generic class implements a generic interface, the type parameters follow both the class and the interface name.
* A generic method defines its own formal type parameters. You can define a generic method in a generic or a non-generic class.
* To define a generic method in a non-generic class or interface, you must define the type parameters with the method in its type parameter section.
* A method’s type parameter list is placed just after its access and non-access modifiers and before its return type. Because a type parameter could be used to define the return type, it should be known before the return type is used.
* You can define a generic method in a generic class or interface, defining its own type parameters.
* You can also define a generic constructor in a generic class.
* You can specify the bounds to restrict the set of types that can be used as type arguments to a generic class, interface, or method. It also enables access to the methods (and variables) defined by the bounds.
* For a bounded type parameter, the bound can be a class, an interface, or an enum, but not an array or a primitive type. All cases use the keyword extends to specify the bound. If the bound is an interface, the implements keyword isn’t used.
* A type parameter can have multiple bounds. The list of bounds consists of one class or multiple interfaces.
* For a type parameter with multiple bounds, the type argument must be a subtype of all bounds.
* The wildcard? Represents an unknown type. You can use it to declare the type of a parameter; a local, instance, or static variable; and a return value of generic types. But you can’t use it as a type argument to invoke a generic method, create a generic class instance, or for a super type.
* You can assign an instance of a subclass, say, String, to a variable of its base class, Object. But you can’t assign ArrayList<String> to a variable of type List<Object>. Inheritance doesn’t apply to type parameters.
* When you use a wildcard to declare your variables or method parameters, you lose the functionality of adding objects to a collection.
* To restrict the types that can be used as arguments in a parameterized type, you can use bounded wildcards.
* In upper-bounded wildcards, the keyword extends is used for both a class and an interface.
* For collections defined using upper-bounded wildcards, you can’t add any objects. You can iterate and read values from such collections.
* You can use final classes in upper-bounded wildcards. Although class X extends String won’t compile, <? extends String> will compile successfully.
* You can restrict the use of type arguments to a type and its super types or base types by using <? super Type>, where Type refers to a class, interface, or enum.
* Type information is erased during the compilation process; this is called type erasure.
* When a generic class is compiled, you don’t get multiple versions of the compiled class files.
* The compiler erases the type information by replacing all type parameters in generic types with Object (for unbounded parameter types) or their bounds (for bounded parameter types).
* The Java compiler might need to create additional methods, referred to as bridge methods, as part of the type erasure process.

*Using type inference*

* If you don’t specify the type of type arguments to instantiate a generic class or invoke a generic method, the Java compiler might be able to infer the argument type by examining the declaration of the generic entity and its invocation. If the type can’t be inferred, you might get a compilation warning, an error, or an exception.
* By throwing an unchecked warning, the compiler states that it can’t ensure type safety. The term *unchecked* refers to operations that might result in violating type safety. This occurs when the compiler doesn’t have enough type information to perform all type checks.
* Starting with Java 7, you can drop the type arguments required to invoke the constructor of a generic class and use a diamond—that is, <>. But an attempt to drop the diamond will result in a compilation warning.
* A Java compiler can’t infer the type parameters by using the diamond in the case of generic methods. It uses the type of the actual arguments passed to the method to infer the type parameters.

*Understanding interoperability of collections using raw types and generic types*

* Raw types exist only for generic types.
* You can assign a parameterized type to its raw type, but the reverse will give a compiler warning.
* When you assign a parameterized type to its raw type, you lose the type information.
* When you mix raw types with generic types, you might get a compiler warning or error or a runtime exception.
* You can assign an object of a subclass to reference a variable of its base class. But this subtyping rule doesn’t work when you assign a collection-of-a-derived-class object to a reference variable of a collection of a base class.
* If you declare a reference variable List<Object> to a list, whatever you assign to the list must be of generic type Object. A subclass of Object isn’t allowed.

*Working with the Collection interface*

* The Collection<E> interface represents a group of objects known as its elements.
* There’s no direct implementation of Collection; no concrete class implements it. It’s extended by more specific interfaces such as Set, List, and Queue.
* This collection is used for maximum generality—to work with methods that can accept objects of, say, Set, List, and Queue.
* All collection classes are generic.
* The Map interface doesn’t extend the core Collection interface.
* The Collection interface implements the Iterable interface, which defines method iterator(), enabling all the concrete implementations to access an Iterator<E> to iterate over all the collection objects.
* The methods of the Collection interface aren’t marked as synchronized.

*Creating and using List, Set, and Deque implementations*

* The List interface models an ordered collection of objects. It returns the objects to you in the order in which you added them. It allows you to store duplicate elements.
* In a List you can control the position where you want to store an element. This is the reason that this interface defines overloaded methods to add, remove, and retrieve elements at a particular position.
* Method listIterator() of List can be used to iterate the complete list or a part of it.
* An ArrayList is a resizable array implementation of the List interface.
* An ArrayList uses the size variable to keep track of the number of elements inserted in it. By default, an element is added to the first available position in the array. But if you add an element to an earlier location, the rest of the list elements are shifted to the right.
* If you remove an element that isn’t the last element in the list, ArrayList shifts the elements to the left.
* An ArrayList maintains a record of its size so that you can’t add elements at arbitrary locations.
* ArrayList’s method remove() sequentially searches the ArrayList to find the target object, using method equals() to compare its elements with the target object.
* If a matching element is found, remove(Object) removes the first occurrence of the match found.
* If you’re adding instances of a user-defined class as elements to an ArrayList, override its method equals() or else its method contains() or remove() might not behave as expected.
* The ArrayList methods clear(), remove(), and removeAll() offer different functionalities. clear() removes all the elements from an ArrayList. Remove (Object) removes the first occurrence of the specified element, and remove(int) removes the element at the specified position. removeAll() removes from an ArrayList all of its elements that are contained in the specified collection.
* A Deque is a double-ended queue, a queue that supports the insertion and deletion of elements at both its ends.
* As a double-ended queue, a Deque can work as both a queue and a stack.
* The Deque interface defines multiple methods to add, remove, and query the existence of elements from both its ends.
* Methods addFirst(), addLast(), offerFirst(), and offerLast() add and remove elements from the top and tail.
* Deque also defines methods push(), pop(), and peek() to add, remove, and query elements at its beginning.
* ArrayDeque and LinkedList implement the Deque interface.
* ArrayDeque is a resizable array implementation of the Deque interface.
* Deque’s method peek() only queries elements, it doesn’t remove them.
* Deque’s method remove() just removes an element.
* Deque’s method poll() returns null when Deque is empty and remove() throws a runtime exception.
* All the insertion methods (add(), addFirst(), addLast(), offer(), offer- First(), offerLast(), and push()) throw a NullPointerException if you try to insert a null element into an ArrayDeque.
* You can iterate over the elements of Deque by using an Iterator, returned by methods iterator() and descendingIterator().
* Class LinkedList implements both the List and Deque interfaces. So it’s a double-linked list implementation of the List and Deque interfaces.
* Unlike ArrayDeque, LinkedList permits addition of null elements.
* A LinkedList is like an ArrayList (ordered by index) but the elements are double-linked to each other. So besides the methods from List, you get a bunch of other methods to add or remove at the beginning and end of this list.
* So it’s a good choice if you need to implement a queue or a stack. A LinkedList is useful when you need fast insertion or deletion, but iteration might be slower than an ArrayList.
* Because a LinkedList implements List, Queue, and Deque, it implements methods from all these interfaces.
* The Set interface models the mathematical Set abstraction.
* The Set interface doesn’t allow duplicate elements and the elements are returned in no particular order.
* To determine the equality of objects, Set uses their method equals(). For two elements, say e1 and e2, if e1.equals(e2) returns true, Set doesn’t add both these elements.
* Set defines methods to add and remove its elements. It also defines methods to query itself for the occurrence of specific objects.
* Class HashSet implements the Set interface. It doesn’t allow the addition of duplicate elements and makes no guarantee to the order of retrieval of its elements.
* HashSet is implemented using a HashMap.
* To store and retrieve its elements, a HashSet uses a hashing method, accessing an object’s hashCode() value to determine the bucket in which it should be stored.
* Method hashCode() doesn’t call method equals().
* Method equals() doesn’t call method hashCode().
* Classes should override their hashCode() methods efficiently to enable collection classes like HashSet to store them in separate buckets.
* A HashSet allows storing of only one null element. All subsequent calls to storing null values are ignored.
* Class HashSet uses hashing algorithms to store, remove, and retrieve its elements. So it offers constant time performance for these operations, assuming that the hash function disperses its elements properly among its buckets.
* A LinkedHashSet offers the benefits of a HashSet combined with a LinkedList. It maintains a double-linked list running through its entries.
* As with a LinkedList, you can retrieve objects from a LinkedHashSet in the order of their insertion.
* Like a HashSet, a LinkedHashSet uses hashing to store and retrieve its elements quickly.
* A LinkedHashSet permits null values.
* LinkedHashSet can be used to create a copy of a Set with the same order as that of the original set.
* LinkedHashSet’s method addAll() accepts a Collection object. So you can add elements of an ArrayList to a LinkedHashSet. The order of insertion of objects from ArrayList to LinkedHashSet is determined by the order of objects returned by ArrayList’s iterator (ArrayList objects can be iterated in the order of their insertion).
* A TreeSet stores all its unique elements in a sorted order. The elements are ordered either on their natural order (achieved by implementing the Comparable interface) or by passing a Comparator while instantiating a TreeSet. If you fail to specify either of these, TreeSet will throw a runtime exception when you try to add an object to it.
* Unlike the other Set implementations like HashSet and LinkedHashSet, which use equals() to compare objects for equality, a TreeSet uses method compareTo() (for the Comparable interface) or compare() (for the Comparator interface) to compare objects for equality and their order.
* If two object instances are equal according to their method equals(), but not according to their method compare() or compareTo(), a Set can exhibit inconsistent behavior.
* Classes Enum and File implement the Comparable interface. The natural order of enum constants is the order in which they’re declared. Classes StringBuffer and StringBuilder don’t implement the Comparable interface.

*Map and its implementations*

* Unlike the other interfaces from the collections framework, like List and Set, the Map interface doesn’t extend the Collection interface.
* A Map defines key-values pairs, where a key can map to a 0 or 1 value.
* Map objects don’t allow the addition of duplicate keys.
* The addition of a null value as a key or value depends on a particular Map implementation. A HashMap and LinkedHashMap allow insertion of null as a key, but TreeMap doesn’t—it throws an exception.
* A HashMap is a hash-based Map that uses the hash value of its key (returned by hashCode()) to store and retrieve keys and their corresponding values. Each key can refer to a 0 or 1 value. The keys of a HashMap aren’t ordered. The Hash- Map methods aren’t synchronized, so they aren’t safe to be used in a multithreaded environment.
* You can create a HashMap by passing its constructor another Map object. Additions of new key-value pairs or deletions of existing key-value pairs in the Map object passed to the constructor aren’t reflected in the newly created HashMap.
* Because a HashMap stores objects as its keys and values, it’s common to see code that stores another collection object (like an ArrayList) as a value in a Map.
* You can call method get() on a HashMap to retrieve the value for a key.
* Methods containsKey() and containsValue() check for the existence of a key or a value in a HashMap, returning a boolean value. Methods get() and containsKey() rely on appropriate overriding of a key’s methods hashCode() and equals().
* Class String and all the wrapper classes override their methods hashCode() and equals(), so they can be correctly used as keys in a HashMap.
* HashMap uses hashing functions to add or retrieve key-value pairs. The key must override both methods equals() and hashCode() so that it can be added to a HashMap and retrieved from it.
* When objects of a class that only overrides method equals() (and not method hashCode()) are used as keys in a HashMap, containsKey() will always return false.
* If you add a key-value pair to a HashMap such that the key already exists in the HashMap, the key’s old value will be replaced with the new value.
* You can add a value with null as the key in a HashMap.
* You can use method remove(key) or clear() to remove one or all key-value pairs of a HashMap.
* Method remove() can return a null value, irrespective of whether the specified key exists in a HashMap. It might return null if matching a key isn’t present in HashMap, or if null is stored as a value for the specified key.
* For a HashMap, methods that query or search a key use the key’s methods hash- Code() and equals().
* Method remove() removes a maximum of one key-value pair from a HashMap. Method clear() clears all the entries of a HashMap. Method remove() accepts a method parameter but clear() doesn’t.
* You can use methods size() and isEmpty() to query a HashMap’s size.
* You can use method putAll() to copy all the mappings from the specified map to a HashMap.
* Method putAll() accepts an argument of type Map. It copies all the mappings from the specified map to the map that calls putAll(). For common keys, the values of map that call putAll() are replaced with the values of the Map object passed to putAll().
* The Map interface defines methods keySet(), values(), and entrySet() to access keys, values, and key-value pairs of a HashMap.
* Method values() returns a Collection object, method keySet() returns a Set object, and method entrySet() returns a Map.Entry object.
* Class HashTable wasn’t a part of the collections framework initially. It was retrofitted to implement the Map interface in Java 2, making it a member of the Java Collection framework. But it’s considered legacy code. It’s roughly equivalent to a HashMap with some differences. The operations of a HashMap aren’t synchronized, whereas the operations of a HashTable are synchronized.
* The LinkedHashMap IS-A HashMap with a predictable iteration order. Like a LinkedList, a LinkedHashMap maintains a double-linked list, which runs through all its entries.
* A LinkedHashMap will always iterate over its elements in their order of insertion.
* A TreeMap is sorted according to the natural ordering of its keys or as defined by a Comparator passed to its constructor.
* TreeMap implements the SortedMap interface. Like HashMap and LinkedHash- Map, the operations of a TreeMap aren’t synchronized, which makes it unsafe to be used in a multithreaded environment.
* The TreeMap performs all key comparisons by using method compareTo() or compare(). Two keys are considered equal by a TreeMap if the key’s method compareTo() or compare() considers them equal.
* When you create a TreeMap object, you should specify how its keys should be ordered. A key might define its natural ordering by implementing the Comparable interface. If it doesn’t you should pass a Comparator object to specify the key’s sort order.
* The set of values that you retrieve from a TreeMap is sorted on its keys and not on its values.
* You can create a TreeMap without passing it a Comparator object or without using keys that implement a Comparable interface. But adding key-value pairs to such a TreeMap will throw a runtime exception, ClassCastException.
* When you pass a Comparator object to TreeMap constructor, the natural order of its keys is ignored.
* Because a TreeMap uses method compare() or compareTo() to determine the equality of its keys, it can access the value associated with a key, even though its key doesn’t override its method equals() or hashCode().

*Using java.util.Comparator and java.lang.Comparable*

* The Comparable interface is used to define the natural order of the objects of the class that implements it.
* Comparable is a generic interface (using T as type parameter) and defines only one method, compareTo(T object), which compares the object to the object passed to it as a method parameter.
* Method compareTo() returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.
* The Comparator interface is used to define the sort order of a collection of objects, without requiring them to implement this interface.
* The Comparator interface defines methods compare() and equals().
* You can pass Comparator to sort methods like Arrays.sort() and Collections.sort().
* A Comparator object is also passed to collection classes like TreeSet and Tree- Map that require ordered elements.
* The Comparator interface is used to specify the sort order for classes that – Don’t define a natural sort order

– Need to work with an alternate sort order – Don’t allow modification to their source code so that natural ordering can be added to them

*Sorting and searching arrays and lists*

* Class Arrays in the collections framework defines multiple methods to sort complete or partial arrays of primitive data types and objects.
* When method Arrays.sort() accepts fromIndex and toIndex values to sort a partial array, the element stored at position fromIndex is sorted, but the element stored at position toIndex isn’t.
* A space has a lower ASCII or Unicode value than lowercase or uppercase letters. When arranged in an ascending order, a String value that starts with a space is placed before the String values that don’t start with a space.
* Class Collections defines method sort() to sort objects of List.
* Classes Arrays and Collections define method binarySearch() to search a sorted array or a List for a matching value using the binary search algorithm.
* The array or List must be sorted according to the natural order of its elements or as specified by Comparator. If you pass this method an unsorted list, the results are undefined. If more than one value matches the target key value to be searched, this method can return any of these values.
* Method binarySearch() returns the index of the search key if it’s contained in the list; otherwise it returns (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or list.size() if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

*Using wrapper classes*

* All the wrapper classes are immutable.
* All the wrapper classes implement the Comparable interface. All these classes define their natural order.
* You can create objects of all the wrapper classes in multiple ways:

– *Assignment*—By assigning a primitive to a wrapper class variable

– *Constructor*—By using wrapper class constructors

– *Static methods*—By calling the static method of wrapper classes, like valueOf()

* All wrapper classes (except Character) define a constructor that accepts a String argument representing the primitive value that needs to be wrapped. Watch out for exam questions that include a call to a no-argument constructor of a wrapper class. None of these classes defines a no-argument constructor.
* To get a primitive data-type value corresponding to a string value, you can use the static utility method parseDataType(), where DataType refers to the type of the return value.
* Wrapper classes Character, Byte, Short, Integer, and Long cache objects with values in the range of –128 to 127. These classes define inner static classes that store objects for the primitive values –128 to 127 in an array. If you request an object of any of these classes, from this range, method valueOf() returns a reference to a predefined object; otherwise, it creates a new object and returns its reference.
* Integer literal values are implicitly converted to Integer objects and decimal literal values are implicitly converted to Double objects.
* The objects of different wrapper classes with the same values aren’t equal.
* When arranged in natural sort order, false precedes true.

*Autoboxing and Unboxing*

* Autoboxing is the automatic conversion of a primitive data type to an object of the corresponding wrapper class (you box the primitive value). Unboxing is the reverse process (you unbox the primitive value).
* Wrapper classes are immutable. Adding a primitive value to a wrapper class variable doesn’t modify the value of the object it refers to. The wrapper class variable is assigned a new object.
* Unboxing a wrapper reference variable, which refers to null, will throw a Null- PointerException.

*String processing*

*Regular expressions*

* You create a custom character class by enclosing a set of characters within square brackets []:
* [fdn] can be used to find an exact match of f, d, or n.
* [^fdn] can be used to find a character that doesn’t match either f, d, or n.
* [a-cA-c] can be used to find an exact match of either a, b, c, A, B, or C.
* You can use these predefined character classes as follows:
  + A dot matches any character (and may or may not match line terminators).
  + \d matches any digit: [0-9].
  + \D matches a nondigit: [^0-9].
  + \s matches a whitespace character: [ (space), \t (tab), \n (new line), \x0B (end of line), \f (form feed), \r (carriage)]
  + \S matches a non-whitespace character: [^\s].
  + \w matches a word character: [a-zA-Z\_0-9].
  + \W matches a nonword character: [^\w].
* To use a regex pattern in Java code that includes a backslash, you must escape the \ by preceding it with another \. The character literal \ has a special meaning; it’s used as an escape character. To use it as a literal, it must be escaped.
* For the exam, you’ll need to know these boundary matchers:
  + \b indicates a word boundary.
  + \B indicates a nonword boundary.
  + ^ indicates the beginning of a line.
  + $ indicates the end of a line.
* The coverage of quantifiers on this exam is limited to the following greedy quantifiers:
  + X? matches X, once or not at all.
  + X\* matches X, zero or more times.
  + X+ matches X, one or more times.
  + X{min,max} matches X, within the specified range.

***Search, parse, and build strings***

* Methods indexOf() and lastIndexOf() differ in the manner that they search a target string—indexOf() searches in increasing position numbers and last- IndexOf() searches backward. Due to this difference, indexOf('a', -100) will search the complete string, but lastindexOf('a', -100) won’t. In a similar manner, because lastIndexOf() searches backwards, lastIndexOf('a', 100) will search the string, but indexOf('a', 0) or indexOf('a', -100) won’t.
* Methods indexOf() and lastIndexOf() don’t throw a compilation error or runtime exception if the search position is negative or greater than the length of this string. If no match is found, they return –1.
* Method contains() searches for an exact match in this string. Because contains() accepts a method parameter of interface CharSequence, you can pass to it both a String and a StringBuilder object.
* Methods subSequence (uppercase *S*) and substring (no uppercase letter) accept int parameters and return a substring of the target string.
* Method substring() defines overloaded versions, which accept one or two int method parameters to specify the start and end positions.
* Method subSequence() defines only one variant, the one that accepts two int method parameters for the start and end positions.
* Methods subSequence() and substring() don’t include the character at the end position in the result String. Also, unlike methods indexOf() and lastIndexOf(), they throw the runtime exception StringIndexOutOfBounds- Exception for invalid start and end positions.
* The name of methods subSequence() and substring() can be used to determine their return type. subSequence() returns CharSequence and substring() returns String.
* Methods split(String regex) and split(String regex, int limit) in class String search for a matching regex pattern and split a String into an array of string values.
* The String array returned by split() doesn’t contain the values that it matches to split the target string.
* You can limit the maximum number of tokens that you want to retrieve by using split(String regex, int limit).
* replace(char oldChar, char newChar) returns a new String resulting from finding and replacing all occurrences of the old character with the new character.
* replace(CharSequence oldVal, CharSequence newVal) returns a new String resulting from finding and replacing each substring of the string that matches the old target sequence with the specified new replacement sequence.
* replaceAll(String regex, String replacement) replaces each substring of the string that matches the given regular expression with the given replacement.
* replaceFirst(String regex, String replacement) replaces the first substring of the string that matches the given regular expression with the given replacement.
* Unlike replace(), replaceAll() doesn’t accept method parameters of type CharSequence. Watch out for the passing of objects of class StringBuilder to replaceAll().
* The combination of the replace, replaceAll, and replaceFirst overloaded methods can be confusing on the exam. Take note of the method parameters that can be passed to each of these methods.
* Scanner can be used to parse and tokenize strings.
  + If no delimiter is specified, a pattern that matches whitespace is used by default for a Scanner object.
  + You can specify a custom delimiter by calling its method useDelimiter() with a regex.
  + Method next() returns an object of type String.
  + Scanner also defines multiple nextXXX methods, where XXX refers to a primitive data type. These methods return the value as the corresponding primitive type.
  + Methods hasNext() and hasNextXxx() only return true or false but don’t advance. Only methods next() and nextXxx() advance in the input.
  + Method findInLine() matches the specified pattern with no regard to delimiters in the input.

***Formatting strings***

* Class java.util.Formatter is an interpreter for printf-style format strings.
* Formatter’s format() is used to format data.
* The format specifier takes the following form: %[argument\_index$][flags][width][.precision]conversion
* A format specification must start with a % sign and end with a conversion character:
  + b for boolean
  + c for char
  + d for int, byte, short, and long
  + f for float and double
  + s for String
* If the number of arguments exceeds the required count, the extra variables are quietly ignored by the compiler and JVM. But if the number of required arguments falls short, the JVM throws a runtime exception.
* The - indicates to left-justify this argument; you must specify width as well. Number flags (only applicable for numbers, conversion chars d and f) are as follows:
  + The + indicates to include a sign (+ or -) with this argument.
  + 0 indicates to pad this argument with zeros. Must specify width as well.
  + , indicates to use locale-specific grouping separators (for example, the comma in 123,456).
  + ( is used to enclose negative numbers in parentheses.
* The flags +, 0, (, and , can be specified only with the numeric specifiers %d and %f. If you try to use them with any other format specifier (%b, %s, or %c), you’ll get a runtime exception.
* Format specifier %b
  + You can pass any type of primitive variable or object reference to specifier %b.
  + If the target argument arg is null, then %b outputs the result as false. If arg is boolean or Boolean, the result is the String returned by String.valueOf(). Otherwise, the result is true.
* Format specifier %c
  + %c outputs the result as a Unicode character.
  + You can pass only literals and variables that can be converted to a Unicode character (char, byte, short, int, Character, Byte, Short, and Integer) to the %c specifier. Passing variables of type boolean, long, float, Boolean, Long, Float, or any other class will throw IllegalFormatConversionException.
* Format specifier %f
  + You can format decimal numbers (float, Float, double, and Double) by using the format specifier %f.
  + By default, %f prints six digits after the decimal. It also rounds off the last digit.
* Format specifier %d
  + You can format integers (byte, short, int, long, Byte, Short, Integer, Long) by using the format specifier %d.
  + If you pass literal values or variables of type float, double, Float, or Double to the format specifier %d, the code will throw a runtime exception.
* Format specifier %s
  + %s outputs the value for a primitive variable. For reference variables, it calls toString() on objects that are not null and outputs null for null values.
  + You can pass any type of primitive variable or object reference to specifier %s.

*Exceptions and assertions*

* A method can throw a more specific exception subclass than the one mentioned in its throws clause, but not a more generic one (superclass).
* A method that declares a checked exception to be thrown might not throw it.
* With Java 7, you can rethrow exceptions with more inclusive type checking.