Exceptions and Preconditions

Preconditions if checking is expensive -- Exceptions are better because they have a stronger spec and are easier to use

Checked are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword

Unchecked are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

A method family contains multiple implementations of same name + parameter types (but not return type!) Which method family? is determined at compile time based on compile-time types E.g., family put(Object key, Object value) or family put(String key, String value) Which implementation within the method family runs, is determined at runtime based on the runtime type of the receiver.

Results (Outputs): Return type of B.m may be replaced by subype in subclass A.m. “covariance” E.g., Number B.m() and Integer A.m() This does not violate expectations of the client! E.g., client: B b; … Number n = b.m(). Client expects a Number. Thus, Integer will work fine No new exceptions. Existing exceptions can be replaced by subtypes Java does allow a subtype return type in an overriding method!

In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Map.getExisting when the key is not in the map CHECKED

Rational.divide when the argument is zero CHECKED

Object.clone when the system is out of memory UNCHECKED

File.load when the file does not exist CHECKED

File.load when there is a disk failure UNCHECKED

Checked exceptions when handling special values, or there is a reasonable action the client can take to recover.

Unchecked exceptions when there is system failures or unrecoverable programming error.

Subtype polymorphism – the ability to use a subclass where a superclass is expected. A java subtype is any class that extends another class (or even implements an interface). A true subtype is not something language specific:

A true subtype can always be substituted for the supertype. "Any property guaranteed by supertype must be guaranteed by subtype (true subtyping)"

A subclass B should be substitutable for its superclass A. I.e., B is a true subtype of A. To ensure that B is substitutable:

B does not remove methods from A and For each B.m that “replaces” A.m, B.m’s specification is stronger than A.m’s specification

Function subtyping -- Rule: A f(B) is a function subtype of C f(D) if A is a subtype of C and B is a supertype of D

Overloading occurs when two or more methods in one class have the same method name but different parameters.

Overriding means having two methods with the same method name and parameters (i.e., method signature). One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided its parent class.

Here are some important facts about Overriding and Overloading:

1). The real object type in the run-time, not the reference variable's type, determines which overridden method is used at runtime. In contrast, reference type determines which overloaded method will be used at compile time.

2). Polymorphism applies to overriding, not to overloading.

3). Overriding is a run-time concept while overloading is a compile-time concept.

In Java, == tests for reference equality. This is the strongest form of equality Equality is an equivalence relation

Reflexive a.equals(a) Symmetric a.equals(b) b.equals(a) Transitive a.equals(b) ∧ b.equals(c) 🡪 a.equals(c)

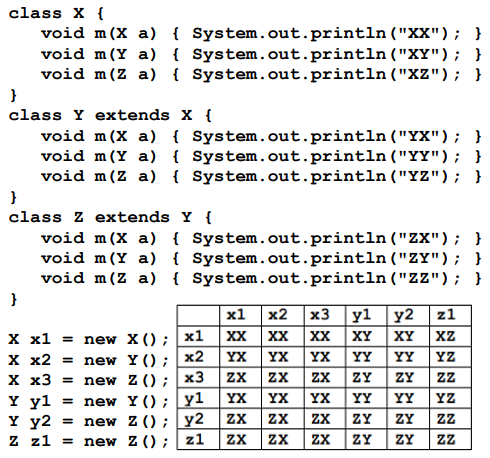
Spec A: @requires x ≥ 0 @returns y such that |y\*y – x| < 0.0001

Spec B: @returns y such that |y\*y – x| < 0.0001 if x ≥ 0, and 0.0 if x < 0

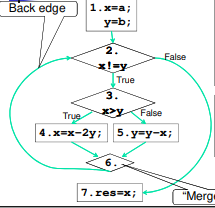
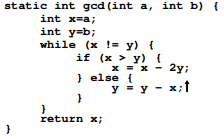
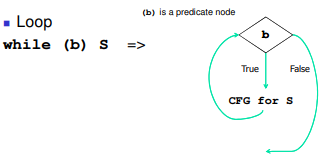
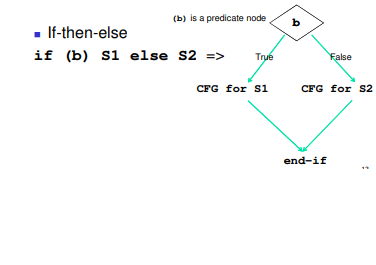
Spec C: @returns y such that |y\*y – x| < 0.0001 if x ≥ 0 @throws IllegalArgumentException if x < 0

Specification A is weaker than B and C, because it guarantees the same as B and C and requires more. B is not clearly or weaker than C. That Being said, the C depicts a square root more accurately (the square root of a -ve number isn't 0, as B details). Therefore, I got C, B, A

a) (FALSE) A true subtype is always a Java subtype. b) (TRUE) In Java, an overriding method can declare a new exception, as long as the new exception is a subtype of one declared in the overridden method. c) (TRUE) The consistency property of hashCode states that for every non-null x and y, x.equals(y) implies x.hashCode() == y.hashCode(). d) (FALSE) In Java, which method family is called, is determined at runtime. e) (TRUE) An important similarity between interface and abstract class is that neither can be instantiated.



If (b) S! else S2 versus While (b) S

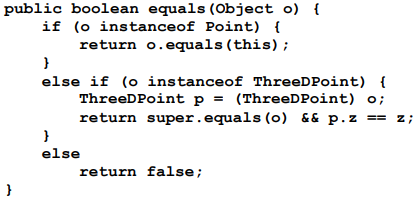


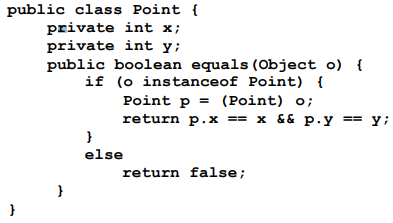
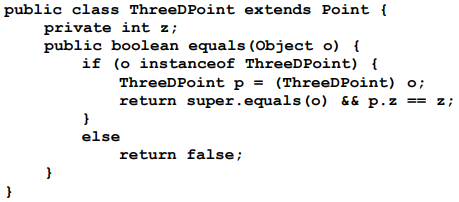
B is a subclass of A. Parameter types of B.m may be replaced by supertypes in subclass A.m. ”contravariance” E.g., B.m(Integer p) and A.m(Number p) This places no extra requirements on the client! E.g., client: B b; … b.m(q). Client knows to provide q a Integer or a subtype of Integer. Thus, client code will work fine with A.m(Number p), which asks for less: an Number or a subtype of Number Java does not allow change of parameter types in an overriding method.

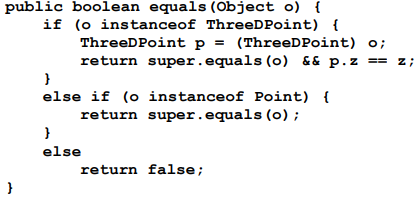
Fill in each box in the above table with the output of the corresponding invocation. For example, fill in the cell for row headed by y1 and column headed by x2 with the output of y1.m(x2).

a) (TRUE/FALSE) It is always possible to cover all def-use pairs in a function. b) (TRUE/FALSE) All-uses coverage implies statement coverage. c) (TRUE/FALSE) A test written against a stronger spec will work with a weaker spec. d) (TRUE/FALSE) Specification tests and black-box tests are different names for the same concept. e) (TRUE/FALSE) A test suite that detects every bug in an implementation, has 100% statement coverage.

a. False, aliasing can be a def-use pair b. Not 100% sure about this one: all-uses => all-defs but not necessarily all statements?

c. False, It's backwards, you cannot substitute a weaker spec for a stronger spec but you can substitute a stronger spec for a weaker one d. True, both are writing tests based on the specs e. False, you don't necessarily need to have covered every statement in order to detect all bugs, i.e. I have a program where I only cover 2/3rds of the statements and the only bugs that occur in the program occur within the 2/3rds that I've covered.



Question 7 is not symmetric because a Point might equal a ThreeDPoint, but the ThreeDPoint will not equal the Point.

Question 8 seems to work fine, but I don't think it would account for ThreeDPoints having different z values (because all ThreeDPoints are Points)

Question 9 is not transitive because two ThreeDPoints with different z's might be equal to the same Point, but are not equal to each other.