Generics and Collections

Monday, February 12, 2018 9:31 AM

Abstract Class

- Can declare fields that are not static/ final

Interface with Default Methods

- Usually used to change interfaces
- Allows for backward compatibility with existing code
  - `default boolean excludes(Point p) { return !contains(p) }`
  - Classes which did not implement new methods can still be compiled (e.g. sort)
- Can only have constant fields (bad practice - constant interface anti-pattern)

Vs. concrete class/ pure interface

Generics

- Type parameter: T, E (treated as variables)
  - Passing type argument to T creates a parameterised type
  - Parametric polymorphism
  - Gives compile time error for incompatible types

Variance of Types

- Invariant: A <: B Generic<A>, Generic<B> has no relationship
  - But all are subtypes of Generic<?> (wildcard type)
- Covariance
  - `class B<T> extends A<T>`

Type Erasure

- Compiler implementation of generic class
- Type parameter replaced with Object (most general type) during compile time
- Bytecode is implemented with backward compatibility
- Implications:
  - Cannot have two methods taking in different parameters for the generic class
    - E.g. foo(Queue<Circle> c) {} and foo(Queue<Point> p) {}
  - Only one copy of static fields (all queues share the same static fields)
  - Does not support static generics
    - `static T y; // cannot decide which type y is`
  - Type casting required
    - `private T[] objects;`
    - `objects = (T[]) new Object[size]; // type cast to access T methods`
- Static generic methods
  - `Static <T> T foo (T t) {}; // T is a type parameter for the method`
  - Scope = within method (T in generic method has no relation to class parameter)
- Cannot create an array of parameterised type
  - Can pass in Point to Queue<Integer>

Type Inference

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

Raw Type

- Queue<Integer> q = new Queue();
- From legacy code; compiler will throw a warning

Wrapper Classes

- Wrap primitive types
- Auto-boxing and Auto-unboxing (type conversion)
    - Not a widening/ narrowing reference conversion: primitive types not subclass of Object

Performance Penalty

- Object requires memory to be allocated/ collected as garbage
- Wrapper classes are immutable (new object created to update value)

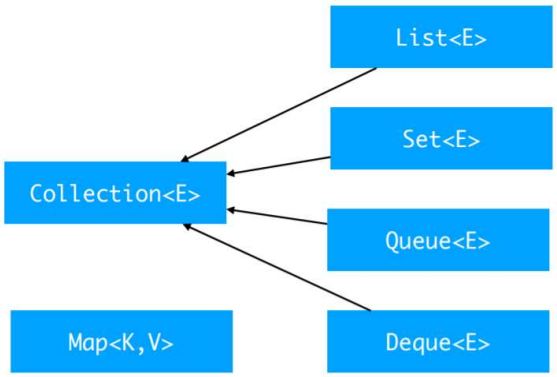
`String and StringBuilder`

- String = immutable
- StringBuilder = mutable (for frequent concatenation)

`.equals()`

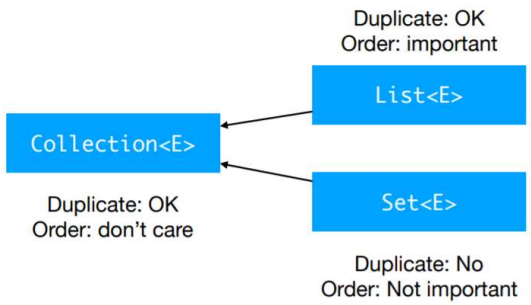
- Autoboxing optimisation
    - Integer caching (returns previously-created objects)
    - Interned string objects maintained for string literals and expressions

Java Collection Framework



Collection<E>

- Generic interface
- `List<E>, Set<E>, Queue<E>, Deque<E>`
    - Deque<E> is a doubly-ended queue



- contains(o) and remove(o) uses equals(o)
- `toArray(); ArrayStoreException`
    - Incorrect type passed in to array

- `addAll(Collection<? extends E> c);`
    - Only accepts subtypes of E (bounded wildcard type parameter), covariant
- `T <: S, X<S> <: X<? super T>`
    - Contravariant
- `UnsupportedOperationException`
    - Implementation decides not to implement one of the operations

Iterator<T> interface

- `for (Iterator<String> I = list.iterator(); i.hasNext(); ) {...}`
- `for (String s : l) {...}`
    - Code improvement: enhanced for loop
    - Cannot change the collection/ remove items during iteration

Comparator<T> interface

- `compare()`
- `list.sort(new StringComparator());`
- In the List<T> interface

Map<K,V>

- Abstract map, hash map…

Exercises

1. B is not abstract and does not override abstract method f() in I
    - Line 2, 8, 9
    - Implementing f(): have to explicitly write public (all interface methods are public)
2. Collections
    - Line 2: List is abstract, cannot be instantiated
    - Line 4: Type inference
3. Generics
    - Line 1: Unexpected type, required reference found int
    - Line 2: Type inference (compiled)
    - Line 3: Raw type (compiled)
4. Integer caching: reference of small numbers maintained (-128 to 127)
    - Always use .equals() to compare reference variables
5. Autoboxing/ unboxing, int to double conversion allowed
6. Autoboxing/ unboxing
    - Part A, B: autoboxing/ unboxing
    - Part C, Line 3: incompatible types: Integer cannot be converted to Double
    - Part D, prints 1.0, 2.0, 3.0
    - Part E, prints 5, 4, 3, 2, 1
7. Autoboxing/ unboxing, Primitive type conversion
    - Part A, B Line 2: incompatible types: possibly lossy conversion from double to int
    - Part C, Line 1: 5.0, Line 2: 2
    - Part D, Line 1: incompatible types: int cannot be converted to java.lang.Double
    - Part D, Line 2: double cannot be converted to Integer
        - Reverse is allowed (see Part F, Line 1)