Garbage collection

What is it ?

1. Unreachable objects cannot affect program execution,
2. Therefore, memory occupied by them can be safely reclaimed,
3. Reclamation process is called garbage collection

Why do it?

1. A running program has a finite amount of memory storage it can use
2. When memory is exhausted, program halts with OutOfMemory exception
3. Want to make most efficient use of memory

Reachable Object

An object is reachable if a reference to it is stored in a local or static variable or it is stored in a field or array element of a reachable object.

At a given point in time, the reachable objects:

1. Are those which can potentially be still used
2. Require space allocated in the heap
3. Cannot be deleted from the heap

Outline of algorithm

1. During execution, unreachable objects are mixed up
2. Must first identify unreachable objects, then we can reclaim them
3. Basic algorithm for this is called “mark and sweep”
4. Reachable objects are “marked” by traversing from object “roots”
5. Could use e.g. depth-first search for this
6. Roots are local variables and static variables
7. – Marked objects are “swept” to the left, unmarked objects are “swept” to the right
8. – Then can reclaim the unmarked objects

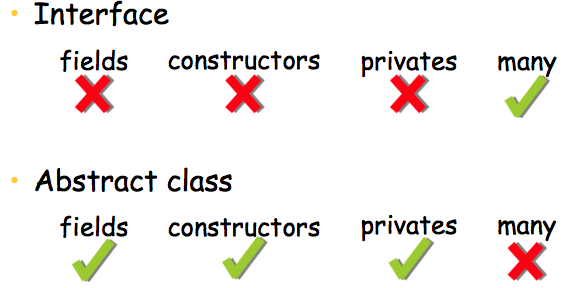
# Java8

**Default and static Interface Methods ?**

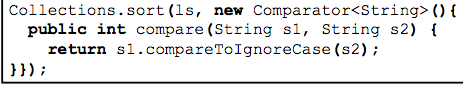
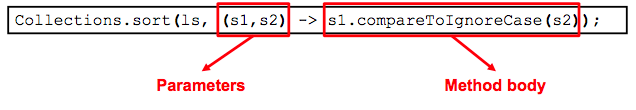
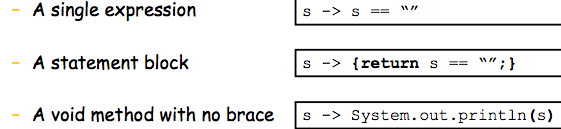
Defa

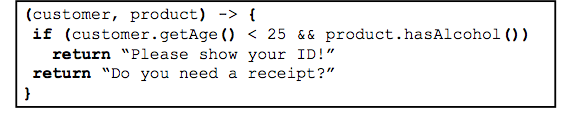
1. Easy to add new methods to interface without changing other classes
2. Two interfaces can have the same default method, but different implementations
3. The common default method(s) must be overridden

Sta

1. Similar to default methods, except that we cannot override them in the implementation classes
2. 

**Lambdas ?**

1. Anonymous single-method class can be unnecessarily long
2. Use Lambdas to make it more compact
3. 
4. 
5. 



1. Anonymous single-method class can be unnecessarily long
2. Use Lambdas to make it more compact
3. – Can omit the data type of the parameters
4. – Can omit the parentheses If only one parameter

**Functional Interfaces?**

1. Interfaces with one and only one abstract method
2. Decorated with **@FunctionalInterface**
3. Can be represented as a lambda expression

**Optional?**

1. Optional<Soundcard> sc = Optional.empty();
2. Soundcard c = new SoundCar();
3. Optional<Soundcard> sc = Optional.of(c);
4. Soundcard c = …;
5. Optional<Soundcard> sc = Optional.ofNullable(c);
6. Optional<Soundcard> sc =..

Sc.isPresent()

Sc.get();

1. •T orElse(T default)
   * Return the default value if the Optional is empty
2. Optional<Soundcard> maybeSoundcard = ...;
3. Soundcard soundcard = maybeSoundcard.orElse(new Soundcard("default"));

**Stream?**

What is stream?

1. • Rich library to query and process collections
2. list.stream()

List<Transaction> transactions = ...

List<Integer> res = transactions

.stream()

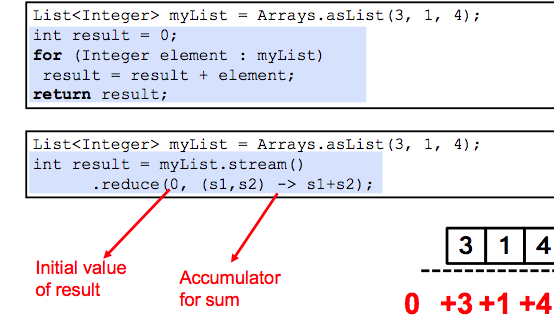
.filter(t -> t.value >= 80)

.sorted((t1,t2) -> t2.value-t1.value)

.map(t -> t.id)

.collect(Collectors.toList());

**Stream Reduce?**



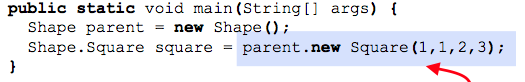
# Nested classes

**Why nested class?**

* 1. Increase logic
  2. Increase encapsulation
  3. More readable and maintainable code

**Non-static?**

Not static – needs a parent object!



**non- static Inner classes**

* 1. have parent pointer

1. –For accessing fields /methods of enclosing class(parent)
2. Parent pointer automatically supplied for new inner class

**• Static Inner Classes**

1. have NO parent pointer!
2. Can NOT access fields/methods of enclosing class
3. Can construct without providing parent pointer
4. If no need to access enclosing info, then this is more convenient (and potentially more efficient).

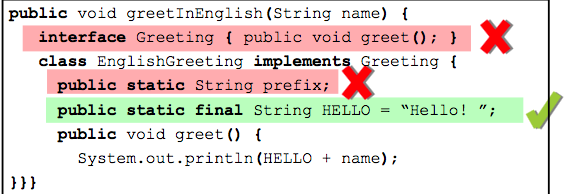
**Non-static inner class vs Static inner class?**

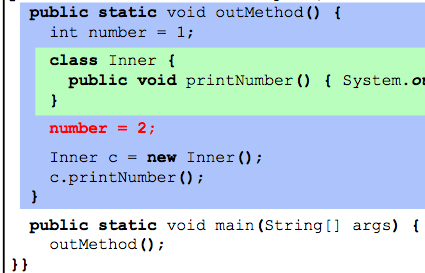
* 1. Inner Class can access members of enclosing class, static cannot
  2. When being constructed, inner class needs to have a parent pointer, static does not need
  3. Inner class cannot have static methods, static can

**Local class?**

Sometimes we want to define classes that are only needed locally

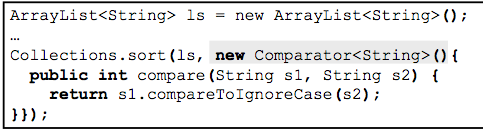
* 1. Can be defined in any block
     1. Method body – for loop
     2. if clause
  2. Can access members of enclosing class
  3. Can access final local variables of enclosing block
  4. Can access effectively final local variables
  5. Cannot have static methods (same as Inner Classes)
  6. Must be non-static, so cannot declare interfaces as local classes
  7. Cannot have static member, unless it’s final primitive (constant)





**Anonymous classes**

* 1. Make code more concise
  2. Declare and instantiate a class at the same time
  3. Local class, but with no name



# Reflection

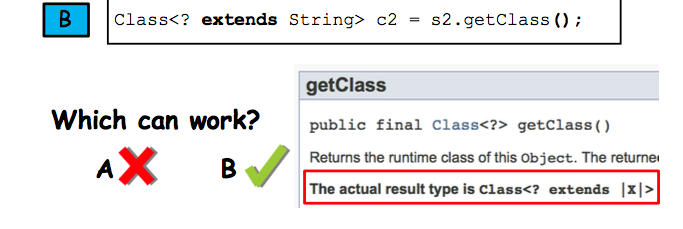
**What is the reflection?**

• Reflection in Java

* + 1. This represents class information
    2. Each object associated with unique instance of Class
    3. Can find out about an object by checking its Class field ( o.getClass() )

**Why is useful?**

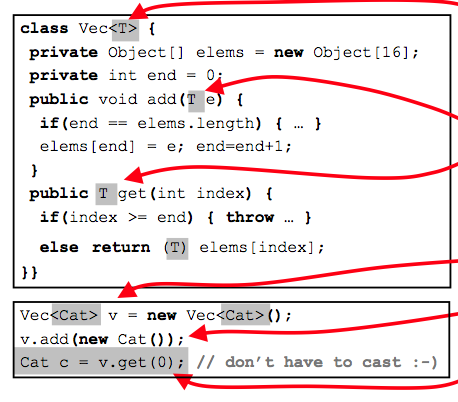
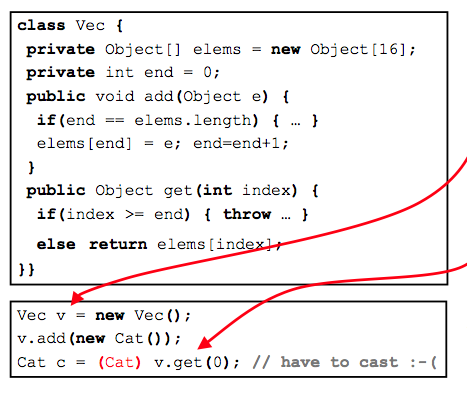
1. Each object associated with unique instance of Class
2. Can find out about an object by checking its Class field ( o.getClass() )
3. Reflection gives access to metadata

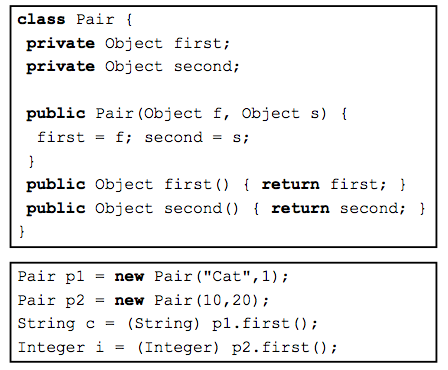
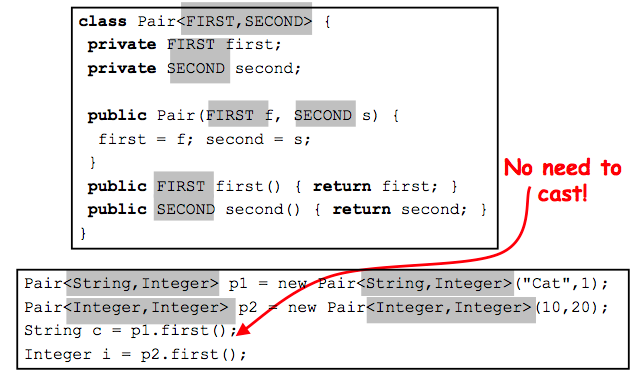


* 1. .class syntax can be used for primitive types
  2. getClass() cannot be used for primitive types
  3. Integer.**class** = Integer.getClass()
  4. Integer.**class** != int.**class**
  5. Integer.**TYPE** = int.**class**
  6. • Singular can access inherited members but not private
  7. • Declared can access private members but not inherited

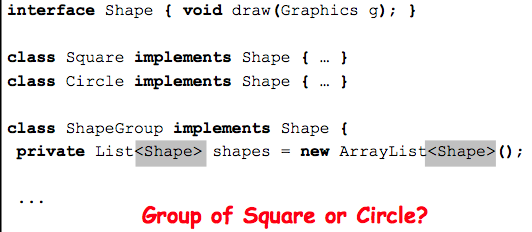
# Generics

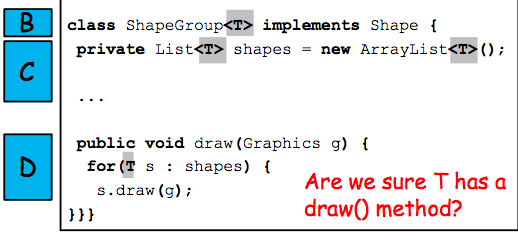
**Generic class?**

****

**** ****

Extend

****

****

**<T extends Shape>**

**<T extends Type> – Type is the name of class or interface**

**<T extends T1 & T2 ...>**

**Generic Methods**

<type parameter> (return type) (method)

• <T> T get(List<T> list, int index) { ... }

• <T extends Comparable> void sort(List<T> list) { ... }

