Basics of programming 3

Reflection, interfaces and lambda

Reflection

Reflection

- Enables runtime inspection of classes
  - extensibility
    - loading external classes in runtime
    - accessing dynamically loaded objects
  - enabling metainformation
    - showing metainformation about classes and objects in runtime
  - debugging and testing
    - accessing and modifying objects in runtime
  - performance issues
    - using reflection results in increased response time

Metainformation about classes

- **`Object.getClass()`**
  - returns the *Class* object representing the class of the object
- **`.class`** access
  - **`boolean.class`** returns an object representing the type *boolean*
  - **`String.class`** returns an object representing class *String*
- **`Class.forName()`**
  - **`Class.forName("java.lang.String")`** return an object representing class *java.lang.**`String`***

Methods of `Class<T>`

- **`Class<? super T> getSuperClass()`**
  - return object representing the superclass
- **`Class<?>[] getClasses()`**
- **`Class<?>[] getDeclaredClasses()`**
- **`Class<?>[] getEnclosingClasses()`**
  - desriptor objects for member/declared/enclosed classes

Methods of `Class<T>` 2

- **`Constructor<T> getConstructor(Class<?>... parameterTypes)`**
  - returns a descriptor object for a public constr. with the specified parameters
- **`Constructor<?>[] getConstructors()`**
  - returns an array of descriptors holding all public constructors
- **`Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)`**
- **`Constructor<?>[] getDeclaredConstructors()`**
  - returns a descriptor(array) for any (private, protected, etc) ctrs

Constructor reflection example

```
package reflect;
public class Person {
   protected String name;
   protected int age;

   public Person(String n, int a) {
      name=n; age = a;
   }
   protected Person(int a) {
      this("John Doe", a);
   }

   public void setName(String n) {name = n;}
   public String getName() {return name;}
}
```

Constructor reflection example

```
Class cls = Person.class;
for (Constructor c : cls.getDeclaredConstructors()) {
   System.out.println(c.getName()
      +": "+c.getParameterCount());
}
Constructor ctr =
   cls.getConstructor(String.class, int.class);
Person p =
   (Person)ctr.newInstance("Eric Cartman", 10);
System.out.println(p.getName());
```

```
reflect.Person: 2
reflect.Person: 1
Eric Cartman
```

Methods of `Class<T>` 3

- **`Field get[Declared]Field(String name)`**
- **`Field[] get[Declared]Fields()`**
  - returns descriptor(array) for field(s)
- **`Method get[Declared]Method(String name, Class<?>... parameterTypes)`**
- **`Method[] get[Declared]Methods()`**
  - returns descriptor(array) for method(s)

Method/field reflection example

```
Class cls = Person.class;
for (Field f : cls.getDeclaredFields()) {
   System.out.println(f);
}
for (Method m : cls.getDeclaredMethods()) {
   System.out.println(m);
}
```

```
protected java.lang.String reflect.Person.name
protected int reflect.Person.age
public java.lang.String reflect.Person.getName()
public void reflect.Person.setName(java.lang.String)
```

Methods of `Class<T>` 4

- **`String getName()`**
  - name of the class
- **`Package getPackage()`**
  - package of the class
- **`int getModifiers()`**
  - modifiers (**`public`**, **`static`**, **`abstract`**, etc) encoded
  - use *Modifier* to decode (*isAbstract(int)*, *isStatic(int)*, etc.)
- **`...`**

Methods of **`Method`**

- **`String getName()`**
  - name
- **`int getModifiers()`**
  - modifiers like for class
- **`Class<?>[] getParameterTypes()`**
  - types of the parameters
- **`Class<?> getReturnType()`**
  - type of return value

Methods of **`Method`** 2

- **`Class<?>[] getExceptionTypes()`**
  - list of possible exceptions
- **`Class<?> getDeclaringClass()`**
  - descriptor of class containing the method
- **`Object invoke(Object obj, Object... args)`**
  - invokes the method
  - on object *obj*
  - with arguments *args*

Method reflection example

```
Person p =
   (Person)ctr.newInstance("Eric Cartman", 10);

Method m = cls.getMethod("setName", String.class);
m.invoke(p, "Eric Theodore Cartman");

System.out.println(p.getName());
```

```
Eric Theodore Cartman
```

Methods of **`Field`**

- **`String getName()`**
  - returns name of attribute
- **`Class<?> getType()`**
  - returns type of attribute as Class object
- **`Object get(Object obj)`**
- **`int getInt(Object obj)`**
  - getting value of attribute (object or primitive value)
- **`void set(Object obj, Object value)`**
- **`void setInt(Object obj, int i)`**
  - setting value of attribute (for an object, value object or primitive)

Method reflection example

```
public class Person {
   protected String name;
   @Unit("year") protected int age;
   ...
```

```
Person p =
   (Person)ctr.newInstance("Eric Cartman", 10);

Field f = cls.getField("age");
f.set(p,11);

System.out.println(
  f.getInt(p)+“, “+
  f.getAnnotation(Unit.class).value());
```

```
11, year
```

Evolving Interfaces

Interfaces revisited

- Interfaces hold method signatures
  - no implementation
  - public access
- Multiple inheritance among interfaces
  - classes can also implement multiple interfaces
- Interfaces help decoupling
  - separate interface and implementation

Interface example and problem

- Let’s define a Stack interface
  - push, pop, top

```
public interface Stack<T> {
   T top();
   T pop();
   void push(T t);
}
```

- Let’s extend Stack with new method
  - dup: duplicates topmost element

```
public interface Stack2<T>
extends Stack<T> {
   void dup();
}
```

Interface extension with default

- What is dup?

```
public void dup() {
   push(top());
}
```

- What to do if *StackImpl* implements *Stack*?
  - let *StackImpl* implement *Stack2*
    - can be used as *Stack* as well
    - must implement new methods
  - lets provide default implementation for *dup*

Interface extension with default

- Default method implementation

```
public interface Stack<T> {
   T top();
   T pop();
   void push(T t);
   default void dup() {
      push(top());
   }
}
```

Default methods

- Advantage
  - no further implementations needed
  - extends legacy interfaces
    - legacy implementations are unaffected
- Disadvantage
  - implementation in interface ☹
  - is it really necessary?

Default method rules

- Extension
  - no mention
    - retains definition in superinterface
  - redeclaration
    - makes method abstract
  - redefinition
    - overrides definition in superinterface
- Implementation
  - default methods can be overridden

Static methods is interfaces

- Interfaces can have static methods

```
public interface X {
   static int foo(String s) {
       return s.length();
   }
}
```

  - only used with interface name
    - e.g. X.foo("hello")
  - useful for defining helper functions

Functional programming in Java

Problem exposition

- Let’s sort Student lists!

```
public class Student {
   private String name;
   private LocalDate birthDate; // since Java8, better ☺
   private double average;
   // + getters-setters

   public Student(String na, LocalDate bd, double a) {
      name = na; birthDate = bd; average  = a;
   }
   public String toString() {
      return name+" "+birthDate.getYear()+", "+average;
   }
}
```

Problem solution

- Classic approach: explicit class

```
public class NComp implements Comparator<Student> {
   public int compare(Student s0, Student s1) {
      return s0.getName().compareTo(s1.getName());
   }
}
```

```
List<Student> l = new ArrayList<Student>();
l.add(new Student("Kenneth McCormick",
             LocalDate.of(1987, 3, 13), 5.0));
l.add(new Student("Kyle Broflovski",
             LocalDate.of(1987, 7, 4), 4.1));
l.add(new Student("Eric Cartman",
             LocalDate.of(1987, 8, 13), 2.3));

Collections.sort(l, new NComp());
```

Problem solution

- Anonymous approach (DON'T)

```
Collections.sort(
   l,
   new Comparator<Student>() {
      public int compare(Student s0, Student s1) {
        return s0.getName().compareTo(s1.getName());
      }
   }
);
```

Problem solution

- Lambda approach (since v8)
  - emerging feature in OO languages
    - originally in functional, declarative programming
    - C++11, C#, python, etc. support some variant

```
Collections.sort(l,
   (s0,s1) -> s0.getName().compareTo(s1.getName())
);
```

Java lambda basics

- *Function interfaces*
  - interface with a single (abstract) method
- Lambda expression
  - in place of objects with function interfaces
  - syntax:
    - `(`*`params`*`) -> (`*`expression`*`)`
    - `(`*`params`*`) -> { `*`function body`*` }`
  - direct access to fields, methods, and local variables of the enclosing scope

Lambda compilation

- Source code

```
void sort(List<T> l, Comparator<T> c);
```

```
Collections.sort(l,
   (s0,s1) -> s0.getName().compareTo(s1.getName())
);
```

- Generated code

```
class LambdaX implements Comparator<Student> {
   public int compare(Student s0, Student s1) {
       return s0.getName().compareTo(s1.getName());
   }
);
```

```
Collections.sort(l, new LambdaX());
```

Lambda compilation

- Source code

```
void sort(List<T> l, Comparator<T> c);
```

```
Collections.sort(l,
   (s0,s1) -> s0.getName().compareTo(s1.getName())
);
```

- Generated code

```
class LambdaX implements Comparator<Student> {
   public int compare(Student s0, Student s1) {
       return s0.getName().compareTo(s1.getName());
   }
);
```

```
Collections.sort(l, new LambdaX());
```

Lambda & Swing: OO

```
final class MyActionListener implements ActionListener {
  JTextField t;
  public MyActionListener(JTextField tt) { t = tt;}
  public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equals("date")) {
      t.setText((new Date()).toString());
    }
 }
}
```

```
...
JTextField t = new JTextField("Type here!");
JButton b = new JButton("Click Me!");
b.setActionCommand("date");
ActionListener al = new MyActionListener(t);
b.addActionListener(al);
...
```

Lambda & Swing: lambda

```
final class MyActionListener implements ActionListener {
  JTextField t;
  public MyActionListener(JTextField tt) { t = tt;}
  public void actionPerformed(ActionEvent ae) {
  ...
```

```
...
JTextField t = new JTextField("Type here!");
JButton b = new JButton("Click Me!");
b.setActionCommand("date");
ActionListener al = new MyActionListener(t);
b.addActionListener(
        ae -> t.setText((new Date()).toString())
);
...
```

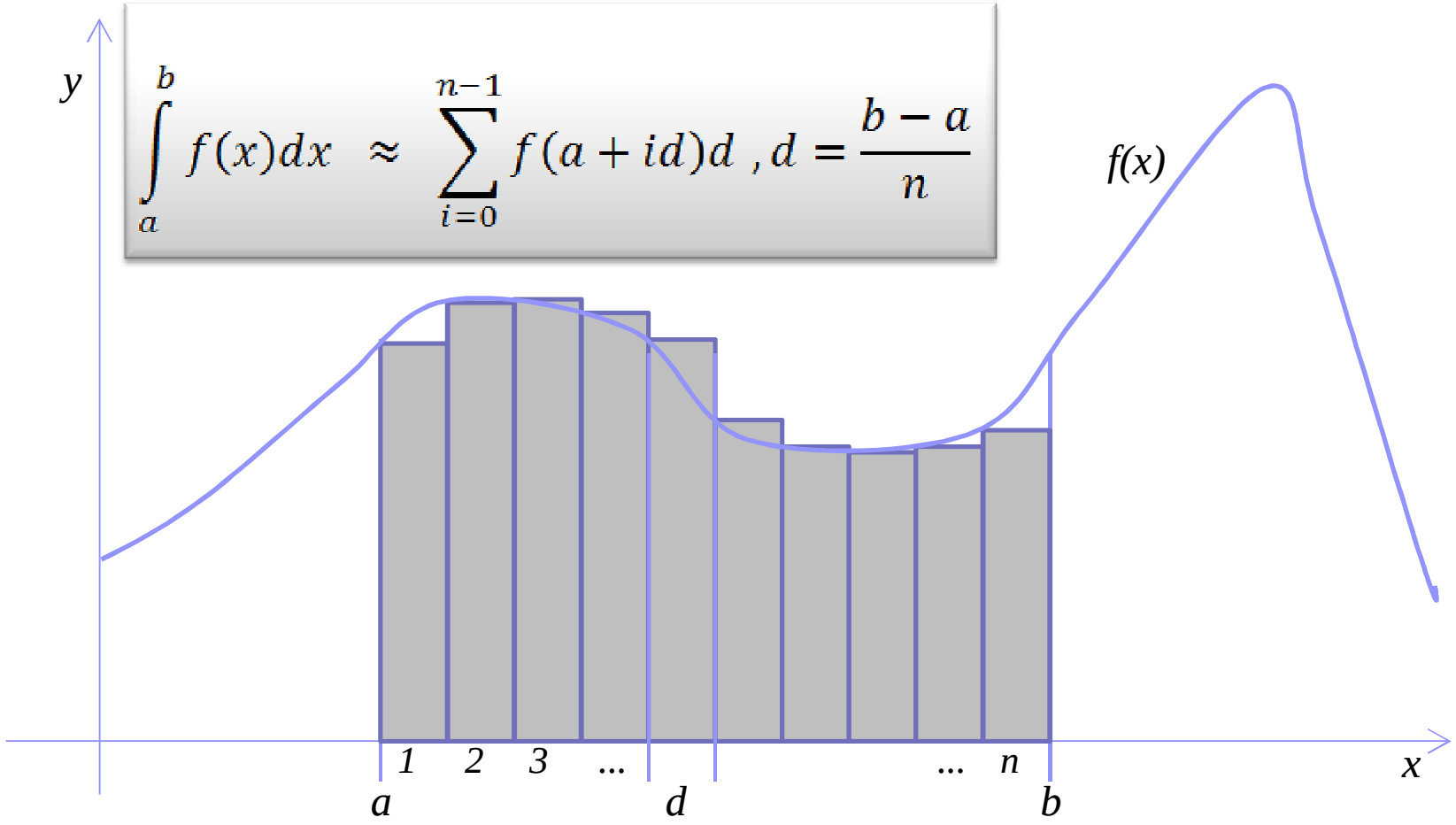
accessing local variable

Lambda and method calls

- Let's do calculus!
  - Calculate definite integrals
    - use rectangle method
  - Functions specified as objects
    - with method *double apply(double d)*
    - cf. C++ functors
  - Interface is needed
    - *Function*

Integral (rectangle method)

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(a+id)d \;, d = \frac{b-a}{n}$$



Integral implementation

```
// general C implementation idea
double sum = 0.0, d=(b-a)/n;
for (double x = a; x < b; x += d) {
   sum += f(x)*d; // area of rectangle at x
}
```

```
// Java implementation, utility class
public class Calculus {
  static double integrate(double a, double b,
                          int n, Function f) {
    double sum = 0.0, d = (b-a)/n;
    for (double x = a; x < b; x += d) {
      sum += f.apply(x)*d;
    }
    return sum;
  }
}
```

method name could be anything

Class or interface expected

Integral with OO approach

```
public interface Function {
   double apply(double x);
}
```

expected interface

```
class Sin implements Function {
   public double apply(double x) {
      return Math.sin(x);
   }
}
```

example implementation

```
Sin s = new Sin();
double result =
   Calculus.integrate(0, Math.PI, 1000, s);
System.out.println(result); // 1.9999983550656881
```

object reference

Using lambda & method ref.

```
double result1 =
   Calculus.integrate(0, Math.PI, 1000,
              x->Math.sin(x)
   );
```

lambda expression

```
double result2 =
   Calculus.integrate(0, Math.PI, 1000,
              Math::sin
   );
```

class method reference

```
double result3 =
   Calculus.integrate(0, Math.PI, 1000,
              s::apply
   );
```

instance method reference (cf. functor)

Function composition

- How to compose functions?
  - *Math.sin* and *Math.abs* separately OK
    - Math::sin, Math::abs
  - *f(x) = abs(sin(x)) ???*

```
public class Compose implements Function {
   Function f1, f2;
   public Compose(Function f1, Function f2) {
      this.f1=f1; this.f2=f2;
   }
   public double apply(double x) {
      return f1.apply(f2.apply(x));
   }
}
```

Function composition example

```
// combination
Compose as = new Compose(Math::abs, Math::sin);
double result4 =
   Calculus.integrate(0,  Math.PI*2, 1000, as);
System.out.println(result4); // 3.999868405188863
```

|sin(x)|

```
// with lambda expression
Compose as2 = new Compose(x->x*x, Math::sin);
double result5 =
   Calculus.integrate(0,  Math.PI*2, 1000, as2);
System.out.println(result5); // 3.141592653589726
```

$\sin^2(x)$

Student sorting revisited

- Sorting with lambda

```
Collections.sort(l,
   (s0,s1) -> s0.getName().compareTo(s1.getName())
);
```

Comparator by lambda expr.

- Sort by any getter method

```
Collections.sort(l,
   Student.comparing(Student::getName)
);
```

Comparator by specifying getter

```
Collections.sort(l,
   Student.comparing(Student::getBirthDate)
);
```

Sorting with any getter

- Student extended...

Works even for *double getAverage()* !

```
public class Student {
   public static
   Comparator<Student> comparing(Getter s) {
     return (Comparator<Student>)
        (o1,o2) -> s.get(o1).compareTo(s.get(o2));
   } // s.get(o) == o.getXXX()
   ...
}
```

calling getter method

```
interface Getter {
   Comparable get(Student s);
}
```

getter method interface

Combining comparators

- Let's sort by name, date, average!
  - current solution works for a single field only ☹
  - cascade sort (when compared fields are equal)

```
public int compare(T o1, T o2) {
   int x = cmp1.compare(o1, o2); // first comparator
   if (x!=0) return x;
   else return cmp2.compare(o1,o2);
                                  // second comparator
}
```

  - building list of comparators is a bonus

Combining comparators

```
public class CComparator<T> implements Comparator<T> {
  Comparator<? super T> cmp1,cmp2;
  public CComparator(Comparator<? super T> c1,
                     Comparator<? super T> c2) {
      cmp1 = c1; cmp2 = c2;
  }
  public int compare(T o1, T o2) {
      int x = cmp1.compare(o1, o2);
      if (x!=0) return x;
      else return cmp2.compare(o1,o2);
  }
  public CComparator<T> then(Comparator<? super T> c){
      return new CComparator(this, c);
  }
}
```

cascade compare

builder method

Combining comparators

```
// sorting on a single field
Collections.sort(l,
          Student.comparing(Student::getAverage));
```

single field

```
// sorting on name, then birthdate, then average
Collections.sort(l,
   new CComparator(
       Student.comparing(Student::getName),
       Student.comparing(Student::getBirthDate)
   ).then(Student.comparing(Student::getAverage)));
```

combinging 2 comptors

adding 3rd one

Generic comparator, step 1

- Let's make comparator generic!

```
public static Comparator<Student>
comparing(Function f) {
   return (Comparator<Student>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

should be generic

generic types

generic comparator

```
public static <T,U> Comparator<T>
comparing(Function<T,U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

generic function (*T*→*U*)

Generic comparator, step 2

- Let's make comparator generic!

```
public static <T,U>
Comparator<T>
comparing(Function<T,U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

U must be *Comparable*

Function may call superclass and return subclass

```
public static <T, U extends Comparable<U>>
Comparator<T>
comparing(Function<? super T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

Generic comparator, step 3

- Let's make comparator generic!

```
public static <T, U extends Comparable<U>>
Comparator<T>
comparing(Function<T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

U may inherit *Comparable*

```
public static <T, U extends Comparable<? super U>>
Comparator<T>
comparing(Function<T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

Generic comparator, final

- Most generic solution

```
public static <T, U extends Comparable<? super U>>
Comparator<T>
comparing(Function<? super T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

  - *T* type to be sorted using a Comparator
  - by field of type *U* that may inherit Comparator impl
  - *Function* is defined in *T* or its superclass
    and selects a field that may be of a subclass of *U*

Lambda and generics in Java 8

- Lambda expects generics
  - functions, comparators, etc
    - whatever the input type, whatever the output type
- Java 8 API
  - interface defaults and statics
  - heavy use of generics
  - lambda enabled

Function interface in Java 8

- `interface java.util.function.Function<T,R>`
  - `R apply(T t)`
    - function body (*f(x)*)
  - `default <V> Function<T,V> andThen(Function<? super R,? extends V> after)`
    - *this*, then *after* (*f(x) = after.apply(this.apply(x))*)
  - `default <V> Function<V,R> compose(Function<? super V,? extends T> before)`
    - *before*, then *this* (*f(x) = this.apply(before.apply(x))*)
  - `static <T> Function<T,T> identity()`
    - a function that returns *x* (*f(x) = x*)

Comparator interface in Java 8

- `interface java.util.Comparator<T>`
  - `int compare(T o1, T o2)`
    original comparator method
  - `static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> getter)`
    returns comparator using getter method (cf. *Student.cmp*)
  - `static <T extends Comparable<? super T>> Comparator<T> naturalOrder() / reverseOrder()`
    returns comparator, *null* values are first/last, otherwise same as *cmp*
  - `default Comparator<T> reversed()`
    returns comparator reversing the result of this comparator

Comparator interface in Java 8

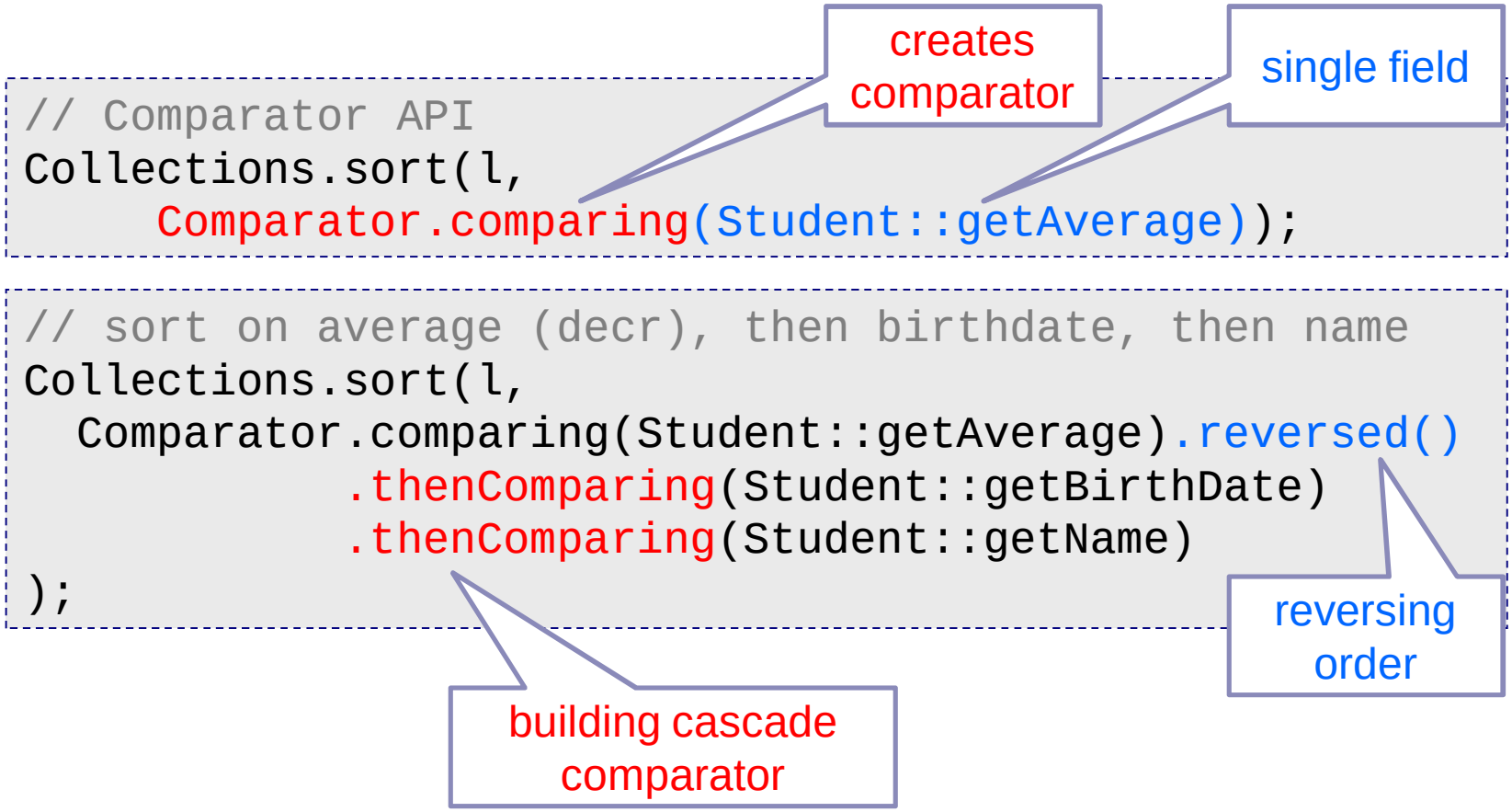
- `default Comparator<T> thenComparing(Comparator<? super T> other)`
- `default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> getter)`
- `default <U> Comparator<T> thenComparing(Function<? super T,? extends U> getter, Comparator<? super U> cmp)`

  returns comparator combining *this* with *cmp* or *getter method*

- `static <T> Comparator<T> nullsFirst(Comparator<? super T> cmp)`
- `static <T> Comparator<T> nullsLast(Comparator<? super T> cmp)`

  returns comparator, *null* values are first/last, otherwise same as *cmp*

Using comparator API



Further features *(selected)*

- Collection
  - `default boolean removeIf(Predicate<? super E> f)`
    - removes each element *x* where *f.test(x)* returns true
- Iterator
  - `void forEach(Consumer<? super T> action)`
    - calls *action.accept(x)* for each element *x*
- Stream
  - e.g. from *Collection.stream()* or *.parallelStream()*
  - unlimited iterator-like object with extra features
    - *distinct, filter, map, forEach, sorted, min, max*, etc.

Example: stream, lambda, etc

- Print names of students
  - with names in range A-L
  - in decreasing order of average marks

```
Collection<Student> l = new List<Student>();
...
l.stream()
  .filter( s -> s.getName().compareTo("M") < 0 )
  .sorted( Comparator.comparing(Student::getAverage)
           .reversed()
         )
  .forEach( System.out::println )
;
```