Victoria University
f Wellington, New Zealan
Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa

SWEN221:
Software Development

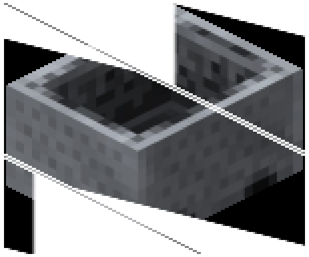
21: Java8: More powerful interfaces!

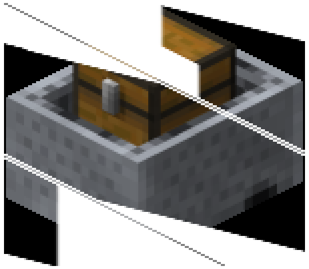
David J. Pearce & Nicholas Cameron & James Noble & Marco Servetto
Engineering and Computer Science, Victoria University

Default methods

- Now interface can contain method implementation!
- Static methods:
  - Works exactly as normal static methods, Convenient to return "predefined" implementations of an interface
- Default methods:
  - A "default" implementation for a method, very similar to an implemented method in an abstract class.

Combining implementations!





```
interface Chest{
  List<Item> get();
  default void depositItem(Item i){/*...*/}
}
interface Minecart{
  Point getPosition();
  void setPosition(Point val);
  default void move(Map map){/*...*/}
}
interface MinecartChest extends Chest,Minecart{
  static MinecartChest factory(Point p){
    return new MinecartChest(){
      Point position=p;
      List<Item> items=new ArrayList<>();
      public List<Item> get() {return items;}
      public Point getPosition() {return this.position;}
      public void setPosition(Point val){this.position=val;}
};  }  }
```

interfaces and abstract classes

- Interfaces:

  🚫 fields 🚫 constructors 🚫 privates many!

- Abstract classes:

  fields constructors privates 🚫 many

- Abstract classes with no fields
  - can you replace it with interface?
  - does it improve code reuse?

Old and new

Comparators using long syntax for anonimus classes

```
Collections.sort(ls,new Comparator<String>(){
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
    }});
```

Comparators using short syntax for anonimus classes

```
Collections.sort(ls,(s1,s2)->s1.compareToIgnoreCase(s2));
```

- Convenient syntax for anonymous nested classes

Extensive use for event handler

```
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    MiniGui g = new MiniGui();
    ...
    JButton b = new JButton("------Bar------");
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("Button pressed");
      }});
    ...}});
```

Before and after

```
SwingUtilities.invokeLater(()->{
  MiniGui g = new MiniGui();
  ...
  JButton b = new JButton("------Bar------");
  b.addActionListener(e->System.out.println("Button pressed"));
  ...});
```

Alternatives for syntax

```
person-> person.getAge()

(p1,p2)-> p1.getAge()>p2.getAge()

()-> System.currentTimeMillis()

(customer,product)-> {
  if(customer.getAge()<25 && product.hasAlcohol()){
    return “Please, show me your id!”
    }
  return “Do you need a receipt?”
  }

person-> person.getAge()
==
(Person person)-> person.getAge()
==
person->{return person.getAge();}
```

Guided exercise

- In this code there is a lot of repetition!
- Use lambdas and factorize the code!

```
public static int sum(List<Integer> list){
  assert !list.isEmpty();//or if(list.isEmpty()){throw...}
  int res=list.get(0);
  for(int i=1;i<list.size();i++){res=res + list.get(i);}
  return res;
  }
public static int mul(List<Integer> list){
  assert !list.isEmpty();//or if(list.isEmpty()){throw...}
  int res=list.get(0);
  for(int i=1;i<list.size();i++){res=res * list.get(i);}
  return res;
  }
```

Guided exercise

- Can we write Reduce.of(list, lambda)?

```java
public static void main(String[] arg){
  List<Integer> list = Arrays.asList(1,2,3,4,5,6,7,3);

  System.out.println(Reduce.of(list,(a,b)->a+b));

  System.out.println(Reduce.of(list,(a,b)->a*b));

  System.out.println(Reduce.of(list,(a,b)->
    {if(a>b){return a;}return b;}));

  }
```

Guided exercise

- Can we write Reduce.of(list, lambda)?

```
public interface Reduce<T> {
  T apply(T e1, T e2);
  public static <T> T of(List<T> list,Reduce<T> fun){
    assert !list.isEmpty();//or if(..){throw..}
    T res=list.get(0);
    for(int i=1;i<list.size();i++){
      res= fun.apply (res, list.get(i));
    }
    return res;
  }
//compare it with the specific code of before:
//assert !list.isEmpty();
//int res=list.get(0);
//for(int i=1;i<list.size();i++){res=res + list.get(i);}
//return res;
```

Syntax and types

- Can use short syntax to implement any ***Functional Interface:***
  - An interface that needs exactly one method implementation to be fully satisfied.
- Examples (Java before 8):

Comparable<T>, Comparator<T>, Runnable, Callable<V>,AutoCloseable

- In Java8, > 40 different functional interfaces:
  - no need to memorize them all!

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.htm

Function in Java8 java.util.function

Main Java 8 functional interface: Function<T,R>

- A function from type T (parameter) to type R (return type)
- Some composition behaviour provided!

```
interface Function<T, R> {

  R apply(T t);//method still to define, often using the new syntax

  static <T>
  Function<T, T> identity(){return t -> t;}

  default <V>
  Function<V, R>compose(Function<? super V, ? extends T> before){
        return (V v) -> apply(before.apply(v));
    }

  default <V>
  Function<T, V> andThen(Function<? super R, ? extends V> after){
        return (T t) -> after.apply(apply(t));
    }
}
```

- Minimal code, but not “simple”

Function in Java8 `java.util.function`

```
Function<Integer,Integer>multiply2=x->x*2;

Function<Integer,Integer>add2=x->x+2;

System.out.println(
  multiply2.andThen(add2).apply(1));//(1*2)+2=4

System.out.println(
  add2.andThen(multiply2).apply(1));//(1+2)*2=6

System.out.println(
  multiply2.andThen(multiply2).apply(1));//1*2*2=4

System.out.println(
  add2.compose(multiply2).apply(1));//(1*2)+2=4
  //==multiply2.andThen(add2)
```

- Simple when sub/super types are not involved

New functional interfaces in Java8

We have seen: `Function`

Now: `Consumer<T>`

- A kind of function that eats up a value.

- Has `accept` method returning void
- Has an `andThen` method to compose `Consumers`:
  values `accepted` by a composed consumer are accepted by both consumers

New functional interfaces in Java8

We have seen: `Function,Consumer`

Now: `Supplier<T>`

- A kind of function that takes no arguments.
- Has a get method returning a value of type T

New functional interfaces in Java8

We have seen: `Function,Consumer,Supplier`

Now: `Predicate<T>`

- A kind of function that takes 1 argument
- Has a `test` method returning a boolean
- Has `and, or, negate` methods allowing to compose `Predicates`.