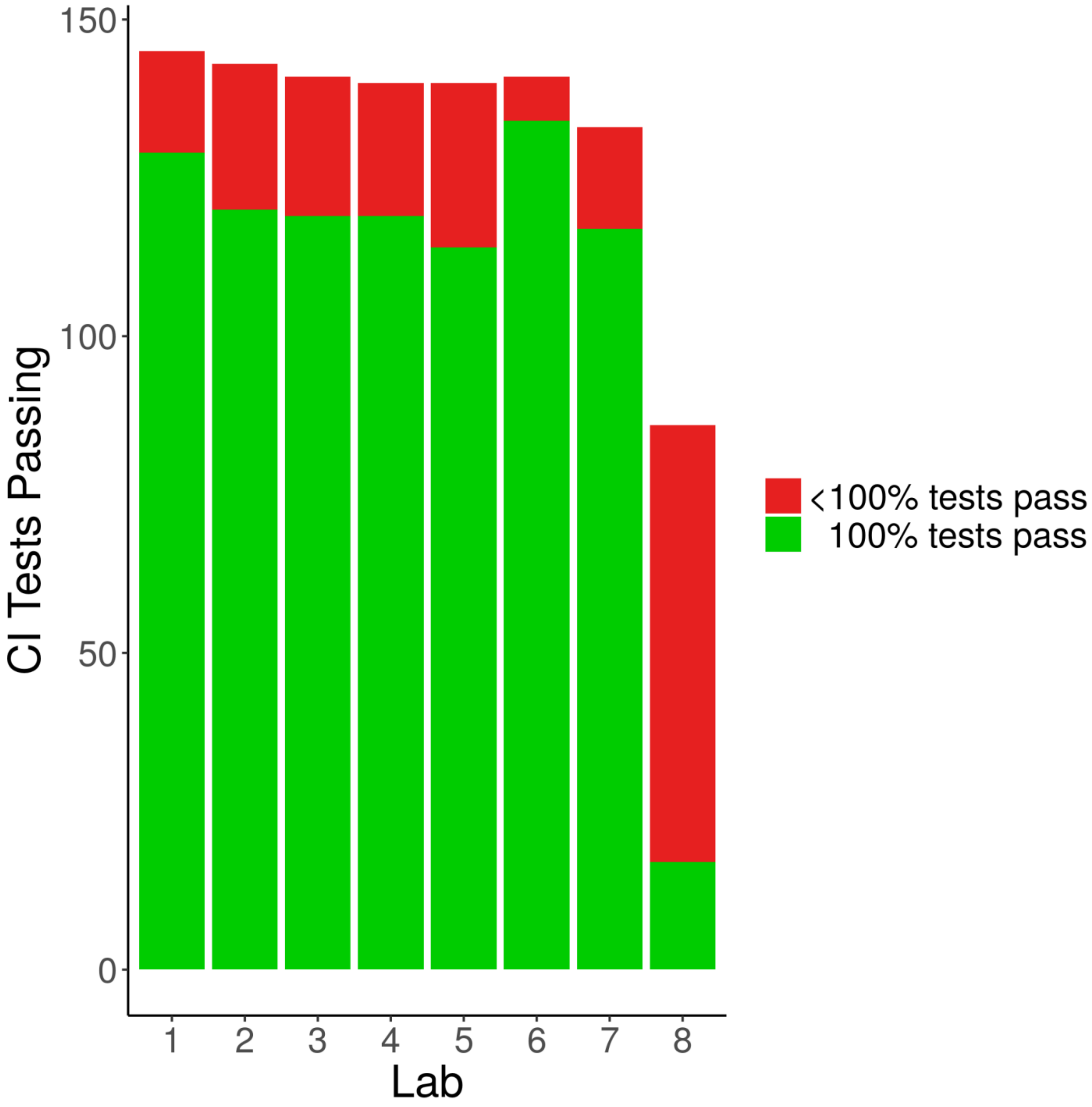
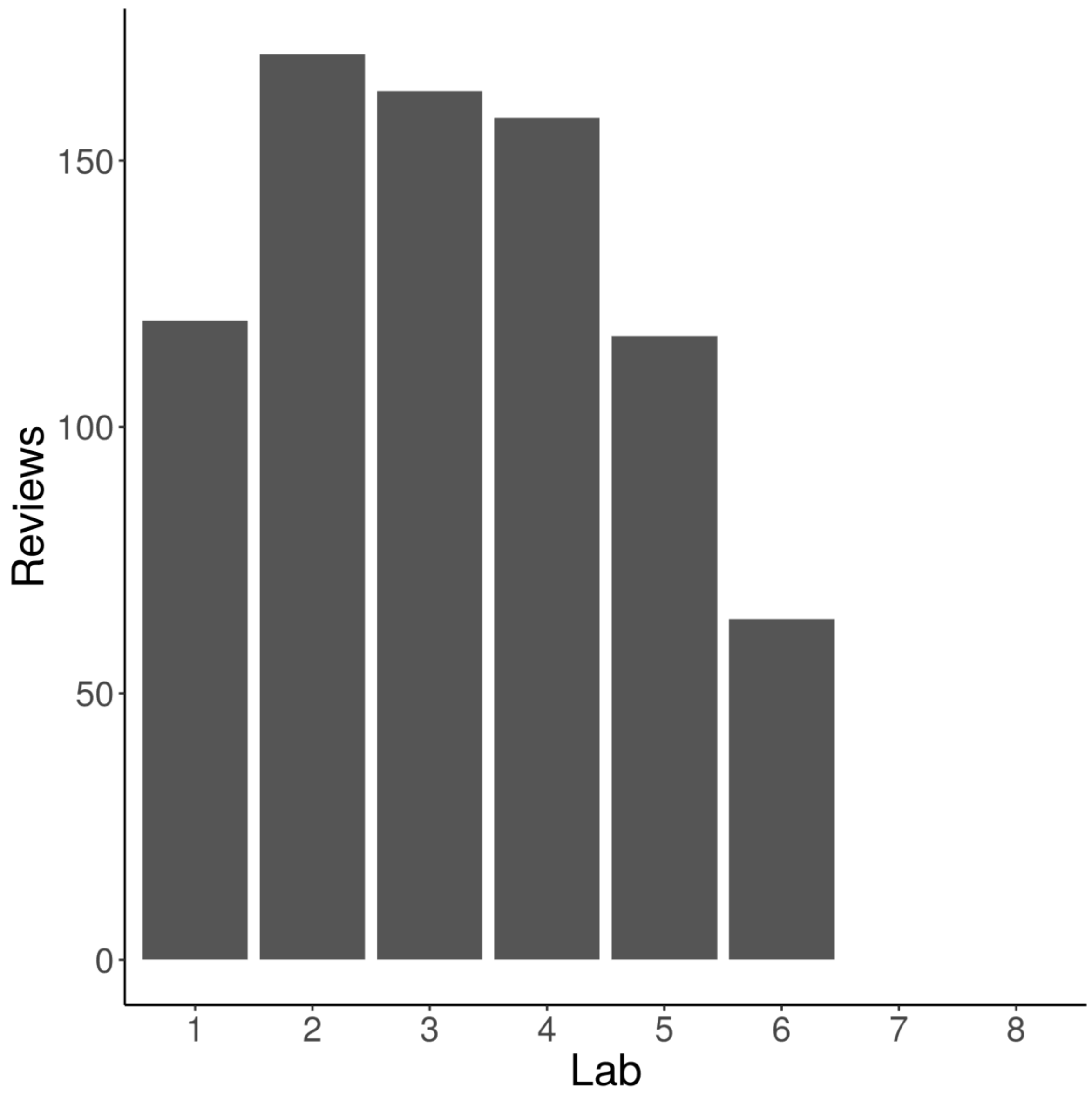
Lab progress





Lab deadlines

- Code reviews labs 1-6
  - Group 1: Friday 15th March (today)
  - Group 2: Monday 18th March

- Lab 8 ***extended deadlines***
  - Group 1: Friday 22nd March
  - Group 2: Monday 25th March
  - (no 5 day late submission)

Code reviews

Remember to **read** reviews about your code!

Code reviews

- Code reviews for labs 7 and 8 will assigned
- I will check your code reviews on GitLab
- I will send out final spreadsheet after that

Upcoming lectures

- Thursday 14th (yesterday)
  - Merge sort and quick sort
- Friday 15th (today)
  - Advanced Java
- Thursday 21st
  - SD3 revision lecture the exam
- Friday 22nd
  - Guest lecture about security implications with Java
    - Dr Manuel Maarek

No Socrative today

Software Development 3 (F27SG)

Lecture 18

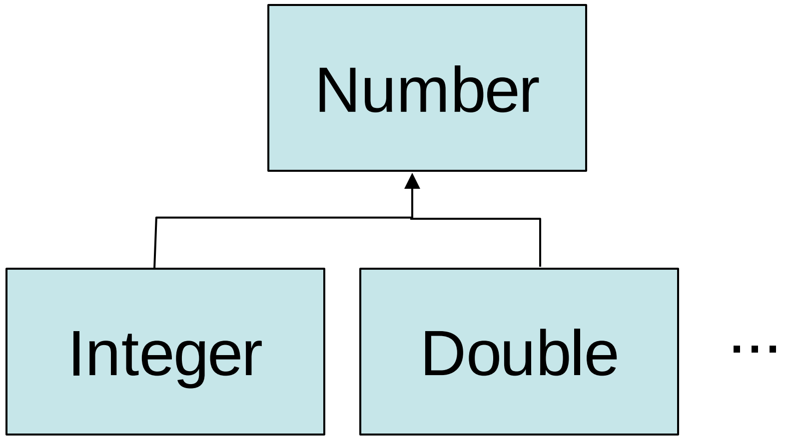
Some Advanced Java Concepts

Rob Stewart

Outline of lecture

- Java Generics
  - Type safe stacks/queues/..
- Comparator interface
  - How to avoid duplicating similar code patterns
- Anonymous classes
  - How to inline comparators
- Lambda expressions
  - How to do *functional programming* in Java

Casting revisited



- Process of converting one type into another type is called **casting**
  - syntax is **(type) expr**
  - often done implicitly in Java

```
double d = 3.2;
int i = (int) d;
double d2 = d / (double) i ;
```

- We can also cast between objects/interfaces in a hierarchy
  - **Widening** converts a type to a *wider* (more general) type

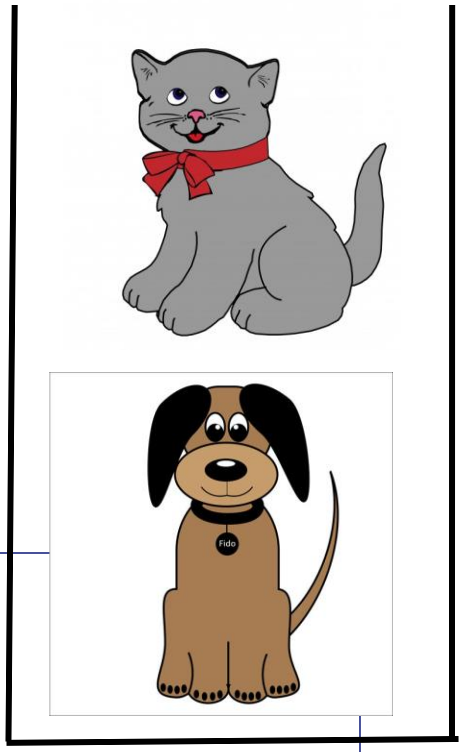
```
Integer i = new Integer(3);
Number n = i;
```

  - **Narrowing** converts a type to a *narrower* (more specific type)

```
Number n = new Integer(3);
i = (Integer) n;
```

A Casting Issue

- We often separate between
  - **static** errors discovered at compile time and in Eclipse
  - **run-time/dynamic** errors discovered while running the program



- Consider this program

```
Stack cats = new Stack();
cats.push(new Dog(“Spot”));
cats.push(new Cat(“Tom”));
Cat cat1 = (Cat) cats.pop(); // this is “Tom the cat”
Cat cat2 = (Cat) cats.pop(); // this is actually “Spot the dog”
```

- Casting error will only be raised at the last line and at **run-time**
- Run-time errors are often much harder to find
- These can easily be missed during testing
- We would like to find as many of the bugs as possible at **compile-time**

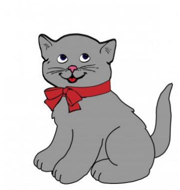
Why did this happen?

- We store type **Object** in the stack every time we pop() an element
  - an Object is returned
    - which we need to cast to the correct type
  - A runtime error is raised if casting is not possible

- This could be overcome
  - by developing a stack to hold **Cat** instances
    - instead of Object instances
  - But requires implementation of new stack for each data type

Java Generics

- Introduce a generic type using angle brackets **<E>**
- When creating a specific object we instantiate this type
  - no casting is required
  - errors are found at compile time

Stack<  **> stack = new Stack<**  **>;**

- We get the **best of both worlds**
  - The static properties of use a specialised type
  - But without having to re-implement our ADT for each type…

- You have already used Generics:

  ArrayList**<String>** list = new ArrayList**<String>**();

Stack Interface in Java

- Remember the Stack interface from lecture 3
- Elements are of type **Object**
  - which is inherited by all other objects

```
public interface StackI {
  public int size();
  public boolean isEmpty();
  public Object top() throws StackException;
  public void push(Object element);
  public Object pop() throws StackException;
}
```

Generics Stacks Demo

- Interface
- Implementation
- Try adding a Dog to a Cat stack

Generic Stack Interface in Java

- We can re-implement our stack using Generics
- Instead of using an **Object,** use a Generic type **<E>**

```
public interface StackI<E> {
   public int size();
   public boolean isEmpty();
   public E top() throws StackException;
   public void push(E element);
   public E pop() throws StackException;
}
```

Stack Implementation using Generics

```
public class Stack<E> implements StackI<E> {
  private int top;
  private int capacity;
  private E[] S;
  private static int MAX = 100;

  public ArrayStack(int capacity){
    this.capacity = capacity;
    S  = (E[]) new Object[capacity];
    top = -1;
  }
  public ArrayStack(){
   this(MAX);
 }
```

Stack Implementation using Generics

```
public void push(E e) throws StackException {
    if (size() == capacity)
      throw new StackException("Stack is full.");
    S[++top] = e;
}


....
```

Use of Generic Stack

- When creating a stack we now need to show in **<brackets>** the type of objects it should hold
- Casting is no longer required:

```
Stack<Cat> cats = new Stack<Cat>();
cats.push(new Cat("Tom"));
Cat cat1 = cats.pop();
```

- If we try to push the wrong type of object the error will be flagged at compile time

```
cats.push(new Dog("Spot"));
```

Exercise

- Recall the Queue interface

```
public interface QueueI {
  public int size();
  public boolean isEmpty();
  public Object front() throws QueueException;
  public void enqueue(Object element);
  public Object dequeue()  throws QueueException;
}
```

1. Provide a Generic version of this interface
2. Create Generic Queue with Dog objects
   – And add a new Dog(“Spot”) to the queue

Solution

1. A Generic Queue interface

```
public interface QueueI<E> {
  public int size();
  public boolean isEmpty();
  public E front() throws QueueException;
  public void enqueue(E element);
  public E dequeue()  throws QueueException;
}
```

2. A Generic queue of Dogs with Spot added:

```
Queue<Dog> queue = new Queue<Dog>();
queue.enqueue(new Dog(“Spot”));
```

Linear Search of Person objects

- We can also generalise our algorithms to use **Generics**
- We will use a variant of **linear search** as illustration
  - Returns **true** if the search succeeds and **false** otherwise
  - takes an array of **Person** objects and a Person object
  - returns **true** if there are any persons of the **same age**

```
public boolean linearSearch(Person[] arr, Person e){
   for(int i = 0; i < arr.length; i++){
      if(arr[i].getAge() == e.getAge())
         return true;
   }
  return false;
}
```

Eclipse demo: linear search

Linear Search of Person objects

- What if we instead wanted to search for Person with the same name? **We have to implement a new version :-(**

```
public boolean linearSearch(Person[] arr,Person e){
   for(int i = 0; i < arr.length; i++){
      if( arr[i].getLastName().equals(e.getLastName()) )
          return true;
   }
  return false;
}
```

```
public boolean linearSearch(Person[] arr, Person e){
   for(int i = 0; i < arr.length; i++){
      if( arr[i].getAge() == e.getAge() )
         return true;
   }
  return false;
}
```

```
public boolean linearSearch(Person[] arr, Person e){
  for(int i = 0; i < arr.length; i++){
    if( /* compare something here */ )
      return true;
  }
  return false;
}
```

What if we instead wanted to search over Strings, integers, ….

Is there a way to generalise this?

Generalising algorithms

- We would like to use **Generics** so that
  - our implementation supports more than one type
  - ... whilst avoiding
    - run-time errors
    - code full of casts
- This requires the ability to **compare** elements
  - smaller, equal or larger

The **Comparator** Interface

- **Comparator** is an **Interface** (in **java.util**) containing one method:

```
public interface Comparator<T>{
  public int compare(T o1, T o2);
}
```

- This method should give a natural order of the objects compared, and returns:
  - **negative** number if **o1 < o2**
  - **0** if they are considered the same
  - **positive** number if **o1 > o2**

A Generic Linear Search

- We can then implement a generic version of linear search
  - Class must be declared with a generic type
  - The specific Person class is replaced by the generic type
  - A comparator is taken as argument to compare elements
- `D[] arr` *// an array of things of type D*
- `D e` *// an instance of class (type) D*
- `Comparator<C>` *// user defined comparator for class D*

```
public class LinearSearchGeneric<D> {

  public boolean linearSearch(D[] arr, D e, Comparator<D> comp){
     for(int i = 0; i < arr.length; i++){
      if(comp.compare(arr[i],e) == 0)
         return true;
     }
    return false;
  }
}
```

Example use of **Comparator**

- To implement a Comparator it has to implement the **Comparator<T>** interface
  - With the generic type **T** instantiated to the correct class
- We can implement our comparison of Person by age as:

```
public  PersonAgeComparator implements Comparator<Person>{
   public int compare(Person p1,Person p2){
         return Integer.compare(p1.getAge(),p2.getAge());   }
}
```

- And by last name as

```
public PersonLastNameComparator implements Comparator<Person>{
   public int compare(Person p1,Person p2){
         return p1.getLastName().compareTo(p2.getLastName());    }
}
```

Using Generic Linear Search

- We can then use our linearSearch method by providing an instance of our comparator interface

```
LinearSearchGeneric<Person> l =
  new  LinearSearchGeneric<Person>();

Person[] ps = new Person[3];
ps[0] = new Person("Harry","Haggis",20);
ps[1] = new Person("Peter","Pie",21);
ps[2] = new Person("Carrie","Curry",19);
Person p = new Person("Frankie","Fish",20);

System.out.println(l.linearSearch(ps,p, new PersonAgeComparator()));
System.out.println(l.linearSearch(ps,p, new PersonLastNameComparator()));
```

Separating **search** from **comparator**

```
public class LinearSearchGeneric<D> {

  public boolean linearSearch(D[] arr,D e, Comparator<D> comp){
      for(int i = 0; i < arr.length; i++){
        if(comp.compare(arr[i],e) == 0)
           return true;
      }
     return false;
  }
}
```

```
LinearSearchGeneric<Person> l =
  new  LinearSearchGeneric<Person>();

Person[] ps = new Person[3];
ps[0] = new Person("Harry","Haggis",20);
ps[1] = new Person("Peter","Pie",21);
ps[2] = new Person("Carrie","Curry",19);
Person p = new Person("Frankie","Fish",20);

System.out.println(l.linearSearch(ps,p, new PersonAgeComparator()));
System.out.println(l.linearSearch(ps,p, new PersonLastNameComparator()));
```

Anonymous classes

- We had to declare a new class

```
public  PersonAgeComparator implements Comparator<Person>{
   public int compare(Person p1,Person p2){
         return Integer.compare(p1.getAge(),p2.getAge());   }
}
```

- Which we only used once

```
System.out.println(l.linearSearch(ps,p, new PersonAgeComparator()));
```

Anonymous classes

- Java allows us to create new classes “on-the-fly” in certain cases
- These are called **anonymous classes**
- Using this we can simplify our code as follows:

```
System.out.println(l.linearSearch(ps,p,
  new Comparator<Person>(){
    public int compare(Person p1,Person p2){
      return Integer.compare(p1.getAge(),p2.getAge());
    }
  }
));
```

Create a new instance of the interface being implemented

Provide implementation of required compare method

Exercise

Show how to call linear search for “C” in the strs array using an **anonymous class:**

```
public class LinearSearchGeneric<D> {
  public boolean linearSearch(D[] arr,D e, Comparator<D> comp){
     … }}
```

```
LinearSearchGeneric<String> l =
  new  LinearSearchGeneric<String>();
String[] strs = ….;
```

```
l.linearSearch(strs,"C", new Comparator<String>(){
    public int compare(String s1,String s2){
        return s1.compareTo(s2);
    }
});
```

Hint1: Use s1.compareTo(s2) to compare String s1 with String s2

Lambda expressions

$$\lambda x.\ E$$

e.g. add two numbers:

$$\lambda x.\ (\lambda y.\ x + y)$$

```
add x y = x + y
add = \x y -> x + y
```

$$\underbrace{\boxed{f}}_{\text{name}} = \underbrace{\boxed{\lambda x\ y}}_{\text{arguments}} . \underbrace{\boxed{x + y}}_{\text{body}}$$

Lambda expressions

- While the code is simpler,
  - anonymous classes are not very readable
- In the latest version of Java (8) **lambda expressions** were introduced
  - This makes the code much more readable

- A lambda expression has the syntax

Argument list	Arrow token	Body
`(int x, int y)`	`->`	`x + y`

Lambda expressions

- Instead of using as anonymous class

```
System.out.println(l.linearSearch(ps,p,
   new Comparator<Person>(){
      public int compare(Person p1,Person p2){
          return Integer.compare(p1.getAge(),p2.getAge());
      }
   }
));
```

- Simplify the code using **lambda expressions**

```
System.out.println(l.linearSearch(ps,p,
   (Person p1, Person p2) ->
              Integer.compare(p1.getAge(),p2.getAge())
));
```

Lambda expressions

Linear search over last name:

```
System.out.println(l.linearSearch(ps,p,
   (Person p1, Person p2) ->
         p1.getLastName().compareTo(p2.getLastName())
));
```

Exercise

Show how to call linear search for “C” in the strs array using a **lambda expression**:

```
public class LinearSearchGeneric<D> {
  public boolean linearSearch(D[] arr,D e, Comparator<D> comp){
     … }}
```

```
LinearSearchGeneric<String> l =
  new  LinearSearchGeneric<String>();
String[] strs = ….;
```

```
l.linearSearch(strs,"C",
   (String s1,String s2) -> s1.compareTo(s2)));
```

Hint1: Use s1.compareTo(s2) to compare String s1 with String s2

Function package

- Java 8 also introduced a function package
- Can define functions using lambda expressions
- Can be treated as values and passed around
  - This is called **higher order programming**
- Examples

Type	Example	Description
Function<F,G>	Function<String,Integer> length = (String s) -> s.length();	A function from type F to type G
Predicate<P>	Predicate<Integer> pos = (Integer i) -> i >= 0;	A predicate (function returning boolean value) of type P
BiPredicate<P,Q>	Predicate<Integer,Ingeger> larger = (Integer I, Integer j) -> i > j;	A binary predicate of types P and Q

Function package

- We can get rid of Comparable and instead use a **BiPredicate**

```
public boolean linearSearchPred(D[] arr,D e, BiPredicate<D,D> pred){
   for(int i = 0; i < arr.length; i++){
     if( pred.test(arr[i],e)  )
       return true;
     }
   return false;
  }
```

Function package

- We can also define a new BiPredicate instead of using it directly
  - (Albeit this is what we wanted to avoid in the first place but this is useful in other cases…)

```
BiPredicate<Person,Person> agepred =
  (Person p1, Person p2) -> p1.getAge() == p2.getAge();

System.out.println(l.linearSearchPred(ps, p, agepred));
```

Summary

- We have given an introduction to more advanced Java features
  - motivated and introduced Java **Generics**
  - Introduced the **Comparator** interface
  - Introduced **anonymous classes**
  - Illustrated **lambda expressions** and the function package

- Optional parts of lab 8 requires these concepts