Objectives

- Review Lambda/Streams Exercise
- Subtype Polymorphism
- Begin talking about Java Concurrency
- Note:
  - Please submit HW 2 until the end of the day

Streams Exercise (1)


```
public static void wordcount(Stream<String> stream) {
    long count = stream.flatMap(s -> Stream.of(s.split(" ")))
                       .count();

    System.out.println(count);
    stream.close();
}
```

- Count of individual words, with a simple split on a space

Streams Exercise (2)


```
public static void filterbywordlength(Stream<String> stream) {
    stream.map(s -> Arrays.asList(Stream.of(s.split(" "))
                        .sorted((x,y) -> y.length() - x.length())
                        .toArray(String[]::new)[0], s))
          .filter(l -> l.get(0).length() > 7)
          .map(l -> l.get(1))
          .forEach(System.out::println);
    stream.close();
}
```

- Create a new stream for each line, sort and collect to an array
- Filter by the longest word on the line
- Return extra copy of original line

Streams Exercise (3)


```
public static void groupwordsbylength(Stream<String> stream) {
    Map<Character, Long> m =
    stream.flatMap((s) -> Stream.of(s.split(" ")))
      .collect(Collectors.groupingBy((s) -> s.toLowerCase()
       .charAt(0),
      Collectors.counting())
);

    System.out.println(m);
    stream.close();
}
```

- Same as first task, but instead use special `groupingBy` collector and specify a downstream counting operation

Type Systems

- Type systems were introduced into programming languages to prevent entire classes of mistakes (at compile time)
  - Passing an `int` when you meant to pass a `boolean`
- Some notable dimensions of type systems:
  - *Strong* $\leftrightarrow$ *Weak*
  - *Static* $\leftrightarrow$ *Dynamic*
  - *Structural* $\leftrightarrow$ *Nominal*
- These dimensions are not always clearly defined, best to think in terms of questions
  - Does type check happen at compile time or run time? (Static-Dynamic)
  - Are there implicit type conversions? (Strong-Weak)
  - Can I change the type of a reference?
  - Do I need to keep track of namings and declarations or what methods (the "shape") objects have?

Type Constructors

- A *type constructor* is a way to create a new type from existing types
- `List` (or something like it) is probably the most familiar type constructor
- Product Types for creating tuples
- Sum Types disjunction or tagged union
- Function Types: the type of functions from one type to another
  - For example, if I have types `A` and `B` then I can form the type of functions f : `A` $\rightarrow$ `B` (note similarity to lambda expressions)

Subtype Polymorphism

- *Polymorphism* allows programmers to treat a value of one type as if it were a value of another type (substitute one for the other)
- *Subtype Polymorphism* substitutability determined by generalization/specialization relationships
  - If `c : Cat` and `Cat extends Animal` then `c` could be substituted whereever an `Animal` is expected
- However, the situation is more complicated when we consider the typing of arrays and collections
  - What is the relationship between `Cat[]` and `Animal[]`, or between `List<Cat>` and `List<Animal>`?

Pair Class with Generics

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
public class Pair<T>
{
  private T first;
  private T second;

  public Pair() { first = null;
                  second = null;}

  public Pair(T first, T second) { this.first = first;
                                   this.second = second; }
  public T getFirst() {return first;}
  public T getSecond() {return second;}

  public void setFirst(T newValue) {first = newValue;}
  public void setSecond(T newValue) {second = newValue;}
}
```

- T is a type variable
- Convention is that the type variable is a single letter
- Class can have more than one type variable

Generic Method


```
public static <T> T getFirst(T... a)
{
   return a[0];
}
```

- Example of a generic method, i.e. a method with type variables
- Add the type parameter to the method call

Bounded Generic Variables


```
public static <T> T min(T[] a)
{
  if (a == null || a.length == 0) return null;
  T smallest = a[0];
  for (int i=1; i < a.length; i++) {
    if (smallest.compareTo(a[i]) > 0) smallest = a[i];
    }
  return smallest;
}
```

- What is a potential problem with this method?

Bounded Generic Variables


```
public static <T extends Comparable> T min(T[] a)
{
  if (a == null || a.length == 0) return null;
  T smallest = a[0];
  for (int i=1; i < a.length; i++) {
    if (smallest.compareTo(a[i]) > 0) smallest = a[i];
    }
  return smallest;
}
```

- Extends in a generic variable declaration can refer either to a superclass or to an interface
- *Bound* for the type variable; separate multiple bounds with an ampersand

Array Example


```
Animal[] arr = new Cat[10]; // ok

arr[0] = new Animal(); // causes runtime exception

// ok
for (int n = 0; n < 10; n++) {
    System.out.println(arr[n]);
}
```

- Recall that *generics* were a later addition to the Java language
- Java arrays were designed to be *covariant*
  - This means, more or less, that `Cat[]` is a subtype of `Animal[]`

Java Generics and Subtyping


```
List<Cat> catList = new ArrayList<Cat>();

// wrong
List<Animal> test = catList;

// correct
List<? extends Animal> animalList = catList;
```

- Parameterized types are *invariant* by default
- In the example here, a `List<Cat>` cannot simply be assigned to a `List<Animal>` reference
  - The *wildcard* makes this happen

Wildcard


```
class C<T> {
  T get();
  void set(T v);
  boolean isSet();
}
// covariant
class C<? extends T> ...

// contravariant
class C<? super T>
```

- The wildcard ? in Java allows programmers to specify the subtyping of parameterized types (by default they are invariant)

Example


```
static void getCovariantList(List<? extends Animal> myList) {
    // Do something with the list
    // myList.add(new Cat());
    Animal a = myList.get(0);
}

static void getContravariantList(List<? super Animal> myList) {
    // Do something with the list
    myList.add(new Cat());
    // Animal a = myList.get(0);
}
```

- Example with `extends` and `super`
- The commented statements would cause a type error in Java

Variance

- *Variance* indicates how subtyping between complicated types works based on the subtyping relationships of the component types
- There are different kinds of variance
  - *Covariance* preserves ordering of types
  - *Contravariance* reverses ordering of types
  - *Invariant* no relationship to the component types
  - *Bivariant* both covariant and contravariant
- Function types give us a case where the variance is naturally contravariant
  - For example, consider the case of the function types (predicates) `Cat` $\rightarrow$ `Boolean` and `Animal` $\rightarrow$ `Boolean`

Functors

- Terminology comes from a branch of mathematics called *Category Theory*
  - All about abstract reasoning with diagrams
- Category Theory is closely related to programming languages, proof theory, and logic via the *Curry-Howard* correspondence
- A *Functor* is a map between categories (maps over the objects of the category and the arrows) that preserves composition
- In our case, we can think of Functors as type constructors that we can map over
  - If $f : \mathtt{A} \to \mathtt{B}$ then, for a functor F we have, $F(f) : F(\mathtt{A}) \to F(\mathtt{B})$
  - This is usually called `map` or `fmap`
  - Example, lists

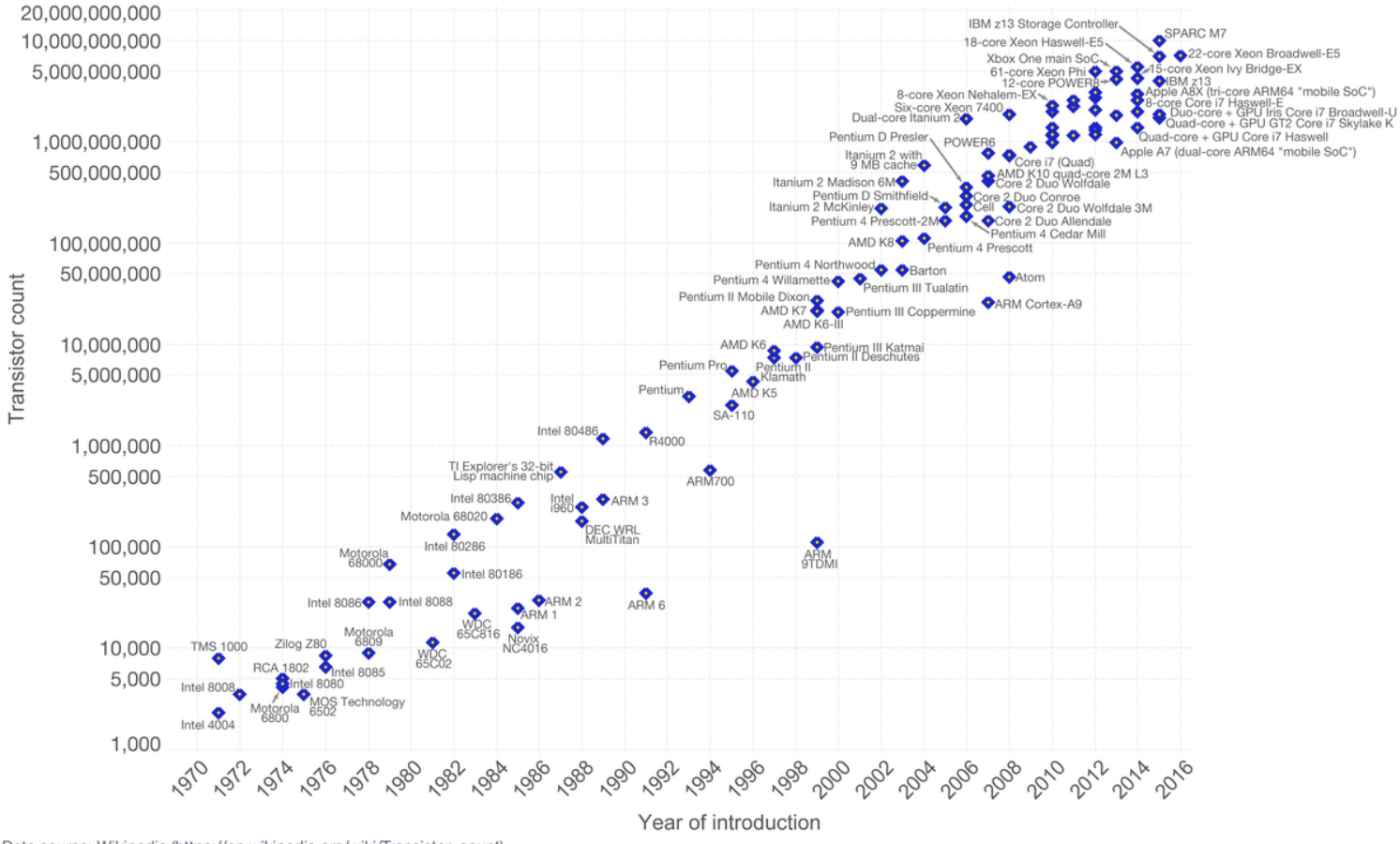
Moore's Law (1)

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Moore's Law (2)

- A current process for semiconductor manufacturing is 14 nanometers (smallest feature that can be machined on the surface of a chip, approximately 1/7000 the width of a human hair)
- Moore's Law is projected to end due to fundamental limitations related to thermal and quantum effects
- However, technology companies have answered with the idea of *multi-core* processors

Concurrency

- *Concurrency* is the composition of processes
- *Parallelism* is the simultaneous execution of processes
- The `java.util.concurrent` package contains many classes and data structures to handle problems with threading and concurrency

Shared State

- A class is *thread-safe* if it behaves correctly when accessed from multiple threads
  - The runtime environment is responsible for scheduling and interleaving of the execution of threads
  - Writing thread-safe code is about dealing with shared, mutable state
  - Most of the time, good object-oriented design principles are consistent with writing good concurrent code

Why care about concurrency?

- To take full advantage of computing resources available today, software engineers should be aware of common challenges caused by multi-threading
- Some applications (such as servlets) are inherently parallel
  - GUIs are inherently asynchronous and need to be responsive to user input

Statelessness


```
public class fact implements Servlet {
  private long count = 0;
  public long getCount() { return count; }

  public void service(ServletRequest req, ServletResponse res) {
    BigInteger i = fromRequest(req);
    BigInteger[] factors = factor(i);

    ++count;
    sendResponse(res, factors);
  }
}
```

- Example from[1], shows a servlet that is not thread-safe due to lack of *atomicity*
- The increment is not a single operation

[1] B. Goetz, T. Peierls, D. Lea, *et al.*, *Java concurrency in practice*. Pearson Education, 2006.

Race Conditions


```
public class InitRace {
  private BigObject o = null;

  public BigObject getInstance() {
    if (o == null) {
      o = new BigObject();
    }
    return o
  }
}
```

- This code illustrates a *race condition*: correctness depends on the relative ordering or timing of threads
- If two threads access the `getInstance` method at the same time they may get two different objects
- Atomic variables are contained in the `java.util.concurrent.atomic` classes

Locks and `synchronized`

- The `synchronized` keyword allows the Java programmer to enforce atomicity
- A `synchronized` block consists of two parts
  - An object that serves as a lock
  - A block of code that is guarded by the lock
- A `synchronized` method simply means a block that spans the entire body of the method
- What would happen if we `synchronized` one of the methods of our servlet, for example `doGet()`