Generics

- Generics introduce type variables/parameters and type arguments.
- Useful for writing reusable code.

Example: `swap( .. )` method

Swap can be implemented as overloads:

```
public static void swap(int[] arr, int i, int j) {
    int tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

public static void swap(double[] arr, int i, int j) {
    double tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

public static void swap(String[] arr, int i, int j) {
    String tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}

public static void swap(Date[] arr, int i, int j) {
    Date tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
}
```

When using these swap methods the data types determine which of the overloads is called:

```
String[] text = "and now for something completely different".split(" ");
swap(text, 3, 5);
int[] xs = new int[]{ 1, 23, 343, 1232 };
swap(xs, 2, 3);
```

We can make a generic method by:

1. Declaring a type-level variable. This variable can take on different type values. We will usually use single-letter capitals, for example: `T`, `R` and `S`.
2. Use the variable in our code where we previously had a data type.

```
public static <T> void swap(T[] arr, int i, int j) {
   T tmp = arr[i];
   arr[i] = arr[j];
   arr[j] = tmp;
}
```

Some syntax:

- `<T>` is the type-variable declaration, like the statement `int x`.
- `T[] arr`, `T tmp` is how the type variable is used.
- Now the calls use the single `swap( .. )` method with different type arguments.

```
String[] text = "and now for something completely different".split(" ");
swap(text, 3, 5);   // with  T = String
int[] xs = new int[]{1,23, 343, 1232};
swap(xs, 2, 3);     // with T = Integer
```

Exercise: write a generic method that counts the number of occurrences of a value in an array.

Sample Solution

```
public static <T> int countOccurrences(T[] arr, T value) {
    int count = 0;
    for(T element : arr) {
        if(element.equals(value))
            count++;
    }
    return count;
}
```

```
String[] text = "and now for something completely different".split(" ");
sout(countOccurrences(text, "and"));   // with T = String
int[] xs = new int[]{1,23, 343, 1232};
sout(countOccurrences(xs, 40));     // with T = Integer
sout(countOccurrences(text, new Date());  // error: T = String and T = Date
```

Example:Homogeneous pairs

Pairs of values of the same type. Ex: `(1, 3)`, `("abc", "def")`, etc…

```
public class Pair<T> {

    private T first;
    private T second;

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

    public void setFirst(T first) {
       this.first = first;
    }

    public T getFirst() {
       return first;
    }

    ....

}
```

Sample usage:

```
Pair<String> p1 = new Pair<String>("abc", "def");  // T = String
sout(p1.getFirst());
Pair<Date> p2 = new Pair<Date>(new Date(), new Date());
sout(p2.getFirst().toString());
```

- Pair<String> say it: ”Pair of String”, List<String> say it: ”List if String”

- The string above is a type argument.
- The type arguement in the constructor is optional: Pair<String> p1 = new Pair<>(”abd”, ”def”);
  - called diamond.
- Note: we can’t use the primitive types as type arguments we need to use their ”object” equivalent:

```
Pair<int> p = new Pair<>(1, 2); // will not compile
Pair<Integer> p = new Pair<>(1, 2);  // will compile
```

Example: Generic stack

We can make a generic stack by ”replacing” the `int` with a type variable `T` (with some adjustments to the constructor):

```
public class Stack<T> {

   private T[] elements;
   private int top;

   public Stack(int capacity) {
      elements = (T[]) new Object[capacity];
      top = 0;
   }

   public void push(T x) {
      if(isFull())
          throw new StackOverflowException();
      elements[top++] = x;
   }

   public T pop() {
      if(isEmpty())
          throw new StackUnderflowException();
      return element[--top];
   }

   public int size() {
      return top;
   }

   public boolean isEmpty() {
      return top == 0;
   }

   public boolean isFull() {
      return top == element.length;
   }
}
```

Example: Heterogeneous pairs

Similar to homogeneous pairs, but now the two values can have different data types:

```
public class HPair<T, S> {
   private T first;
   private S second;

   public Pair(T first, S second) {
      this.first = first;
      this.second = second;
   }

   public T getFirst() {
      return first;
   }

   public void setFirst(T first) {
      this.first = first;
   }

   public S getSecond() {
       return second;
   }
    ...
}
#+begin

HPair<String, Date>  p4 = new HPair<>("abc", new Date());
```

```
HPair<Integer, Character> p5 = new HPair<>(123, 'c');
```

Example: limits of generics

- What about a maximum of a array? Not all types `T` are comparable. We need to add a "bound":

```
public static <T extends Comparable> T max(T[] arr, int size) {
    if(size == 0) throw ...
    T m = arr[0];
    for(int i = 1; i < size; i++) {
       if(arr[i].compareTo(m) > 0)
          m = arr[i];
    }
    return m;
}
```

The bound means that the only types allowed when calling `max( .. )` are those that implement the `Comparable` interface and therefore have an implementation of `compareTo( .. )`