Why Generics

Before generics:

```
List myStrings = new ArrayList();
...
String s = (String)myStrings.getFirst();
s.length();
```

With generics:

```
List<String> myStrings = new ArrayList<String>();
...
String s = myStrings.getFirst();
s.length();
```

- Replaces runtime type checks with compile-time checks
- Should not use raw types in your programs now.

Issue of Covariance

Arrays are covariant

```
sum(Number[] nums){
   nums[0] = new Double(13);
   ...
}
Integer[] b = new Integer[]();
sum(b);   // Compiles, but ArrayStoreException
```

Generics are invariant:

```
sum(List<Number> nums){ ... }
List<Integer> b = new ArrayList<Integer>();
sum(b);   // Incompatible types
```

- Illegal to create an array of a generic type, a parameterized type or a type parameter. E.g. `new List<E>[]`, `new List<String>[]`, `new E[]`
- Non-reifiable types: types where information has been removed at compile-time by type erasure.

Example: Generic Linked List

```
public class MyLinkedList<E>{

  static private class Cell<V> { // generic because static
    private V element;
    private Cell<V> next;
    private Cell(V e) {
      element = e;
    }
  }
private Cell<E> head;
private int size;
```

- **static** nested classes cannot access generic parameters of the outer class.

```
public MyLinkedList () {
}
public MyLinkedList (E[] a) { // construct a list from an array
  this();
  for (int i=a.length-1; i>=0; i--)
    addFirst(a[i]);
}
public MyLinkedList (Iterable<? extends E> c) {
  this();
  MyLinkedList<E> l = new MyLinkedList<E>();
  for (E e : c)
    l.addFirst(e);
  while (!l.isEmpty())
    this.addFirst(l.removeFirst());
}
```

- cannot use new `for` loop on array (iteration in reverse order)
- instance of a generic class created with `new MyLinkedList<E>()`
- can also use `new MyLinkedList<>()` when the generic type can be **inferred** by the compiler (Java 7)
- `<? extends E>` denotes a **subtype** of `E`

```
public MyLinkedList (Iterable<E> c) {
   this();
   MyLinkedList<E> l = new MyLinkedList<E>();
   for (E e : c)
     l.addFirst(e);
   while (!l.isEmpty())
     this.addFirst(l.removeFirst());
 }

Iterable<Integer> c = new java.util.LinkedList<Integer>();
MyLinkedList<Integer> l1 = new MyLinkedList<Integer>(c);

MyLinkedList<Number> l2 = new MyLinkedList<Number>(c);  // won't compile
```

- `Iterable<Integer>` **is not** a subtype of `Iterable<Number>`
- `Iterable<Integer>` **is** a subtype of `Iterable<? extends Number>`
- so the constructor should be
  `public MyLinkedList (Iterable<? extends E> c)`

```
public int size () {
  return size;
}
public boolean isEmpty () {
  return size == 0;
}
public void clear () {
  head = null;
  size = 0;
}
public E getFirst () {
  if (size == 0)
    throw new IllegalStateException("empty list");
  return head.element;
}
```

- note type `E` in signature of `getFirst`
- note use of `IllegalStateException`

```
public E removeFirst () {
  if (size == 0)
    throw new IllegalStateException("empty list");
  E e = head.element;
  head = head.next;
  size--;
  return e;
}
public E get (int n) {
  if (n < 0)
    throw new IllegalArgumentException("negative index");
  if (n >= size)
    throw new IllegalStateException("short list");
  Cell<E> c = head;
  for (int i=0; i<n; i++)
    c = c.next;
  return c.element;
}
```

- generic `Cell<V>` instantiated into `Cell<E>`
- note use of `IllegalArgumentException`

```
public E remove (int n) {
  if (n < 0)
    throw new IllegalArgumentException("negative index");
  if (n >= size)
    throw new IllegalStateException("short list");
  if (n == 0)
    return removeFirst();
  Cell<E> c = head;
  for (int i=1; i<n; i++)
    c = c.next;
  E e = c.next.element;
  c.next = c.next.next;
  size--;
  return e;
}
public void addFirst (E e) {
  Cell<E> c = new Cell<E>(e);
  c.next = head;
  head = c;
  size++;
}
```

```
public void add (E e, int n) {
  if (n < 0)
    throw new IllegalArgumentException("negative index");
  if (n > size)
    throw new IllegalStateException("short list");
  if (n == 0) {
    addFirst(e);
    return;
  }
  Cell<E> c = head;
  for (int i=1; i<n; i++)
    c = c.next;
  Cell<E> d = new Cell<E>(e);
  d.next = c.next;
  c.next = d;
  size++;
}
```

```
public static <T> MyLinkedList<T> fill (T value, int n) {
  if (n < 0)
    throw new IllegalArgumentException("negative length");
  MyLinkedList<T> l = new MyLinkedList<T>();
  for (int i=0; i<n; i++)
    l.addFirst(value);
  return l;
}
```

- generic **method** parameterized with type `T`
- type `T` independent from class parameter `E`
- generic methods can belong to generic or non-generic classes

```
public E[] toArray () {
  E[] a = (E[])new Object[size];
  Cell<E> c = head;
  for (int i=0; i<size; i++) {
    a[i] = c.element;
    c = c.next;
  }
  return a;
}
```

- cannot create a new array of `E`
- **typecast** `(E[])` produces a warning (unchecked cast) unless **suppressed**
- this method is **useless**:

```
MyLinkedList<String> l = new MyLinkedList<>(...);
String[] a = l.toArray();
```

  can be compiled but throws a **ClassCastException**

```
public Object[] toObjectArray () {
  Object[] a = new Object[size];
  Cell<E> c = head;
  for (int i=0; i<size; i++) {
    a[i] = c.element;
    c = c.next;
  }
  return a;
}

@SuppressWarnings("unchecked")
public static <T> void sort(MyLinkedList<T> l, java.util.Comparator<? super T> cmp){
  T[] a = (T[])l.toObjectArray();
  java.util.Arrays.sort(a,cmp);
  l.clear();
  for (int i=a.length-1; i>=0; i--)
    l.addFirst(a[i]);
}
```

- a method `E[] toArray (E[] a)` can be written using **reflection** (it uses the existing array or reallocates a larger one using reflection)
- `<? super T>` denotes a **supertype** of `T`
- `sort` can be called with a list of `Square` and a `Shape` comparator

Why do we need all these <? extends ...> and <? super ...>?

```
interface Sale {
  public int value ();
}
class BookSale implements Sale {...}
class DVDSale implements Sale {...}
```

```
public int totalSales (List<Sale> sales) {...}
```

this method **cannot** be called with a `List<BookSale>` object

Why do we need all these <? extends ...> and <? super ...>?

```
interface Sale {
  public int value ();
}
class BookSale implements Sale {...}
class DVDSale implements Sale {...}
```

```
public int totalSales (List<Sale> sales) {...}
```

this method **cannot** be called with a `List<BookSale>` object

```
public void someMethod (List<Sale> sales) {
  sales.add(new DVDSale(...));
}
```

```
public int totalSales (List<? extends Sale> sales) {
  int sum = 0;
  for (Sale sale : sales)
    sum += sale.value();
  return sum;
}

public void addBookSales (List<? super BookSale> sales) {
  BookSale[] localSales = getBookSales(); // private method
  for (BookSale sale : localSales)
    sales.add(sale);
}
public void addDVDSales (List<? super DVDSale> sales) {
  DVDSale[] localSales = getDVDSales(); // private method
  for (DVDSale sale : localSales)
    sales.add(sale);
}
```

- `totalSales` can be called with `List<Sale>`, `List<BookSale>`, `List<DVDSale>`, `List<? extends Sale>` objects
- `addBookSales` can be called with `List<Sale>`, `List<BookSale>`, `List<? super Sale>` objects
- `addDVDSales` can be called with `List<Sale>`, `List<DVDSale>`, `List<? extends Sale>` objects

Note: arrays are handled differently

```
public void someMethod (Sale[] sales) {
  sales[0] = new BookSale();
}
```

- this method works fine with `Sale[]` **and** `BookSale[]` objects
- it **can** be called with a `DVDSale[]` object
  but will throw an **ArrayStoreException**

Summary

- generics allow for **parameterized types**
- `List<Integer>` is **NOT** a subtype of `List<Number>`
  (`Integer[]` *is* a subtype of `Number[]` but `ArrayStoreException`)
- `<? extends T>` denotes a subtype of `T`
- `<S extends T>` denotes a subtype `S` of `T`
- `<? super T>` denotes a supertype of `T`
- `List<Integer>` **is** a subtype of `List<? extends Number>`
- `List<Number>` **is** a subtype of `List<? super Integer>`
- **cannot** create arrays of a generic parameter
- **static** methods/classes **cannot** access a class' generic parameter
- `Stack<String>` and `Stack<Number>` are the **same** class and share **static** methods and attributes (and have the same `Class` object)