Generics

IRS W-9 Form
Generics

Generic class and methods.
BNF notation
Syntax

Non-parametrized class:

```
<class declaration> ::= 
  "class" <identifier> 
  ["extends" <type>] 
  ["implements" <type list>] 
  "{" <class body> "}" 
```

Generic class

```
<class declaration> ::= 
  "class" <identifier> [<type parameter list>] 
  ["extends" <type>] 
  ["implements" <type list>] 
  "{" <class body> "}" 
```
Syntax

Non-parametrized method

\[
\text{<method declaration> ::= <type> <identifier> "(" [<formal parameters list>] ")" ["throws" <qualified identifier list>] 
"{" <method body> "}"
\]

Generic method

\[
\text{<method declaration> ::= [<type parameter list>] <type> <identifier> "(" [<formal parameters list>] ")" ["throws" <qualified identifier list>] 
"{" <method body> "}"
\]
Syntax

Generic type parameters

<type parameter list> ::= 
    "<" <type parameter> {""," <type parameter>} ">"
<type parameter> ::= 
    <identifier> [ "extends" <bound> ]
<bound> ::= <type> { "&" <type> }

$type argument list$ ::= 
    "<" <type argument> {""," <type argument>} ">"
<type argument> ::= <type>
    | "?" ["extends" <type>]
    | "?" ["super" <type>]
Syntax

\[
\text{<type> ::= <identifier> [<type argument list>]}
\]
\[
\{\ "\ " <identifier> [<type argument list>] \}
\]
\[
\{\ "[\ [\]\]\}\]
\]

/ \text{<primitive type>}

Already we have used ArrayList and LinkedList.

```java
List<BigDecimal> list1 = new ArrayList<>();
list1.add(new BigDecimal("123");
list1.add(new BigDecimal("4567");
System.out.println(list1);

List<String> list2 = new LinkedList<String>();
list2.add("How");
list2.add("now");
list2.add("brown");
list2.add("cow");
System.out.println(list2);
```
Consider building a class to hold two values.

```java
class Pair {
    int first, second;
}
```
Not very flexible. So we try using subtype polymorphism (object-oriented programming). We take advantage of the fact that every object is a subclass of Object.

```java
class Pair {
    Object first, second;
    void setFirst (Object first) { this.first=first; }
    Object getFirst () { return first; }
}

Pair p = new Pair();
p.setFirst (new Object());
p.setFirst ("hello");
p.setFirst (new Integer (5));
System.out.println (p.getFirst());
```
Why Not Subtype?

Subtype polymorphism here is unsafe. The type of the objects is “laundered.” We can put it in; but we can’t take it out as the same type of thing we put in. The class can be abused and this is not discovered until runtime.

class Pair {
    Object first, second;
    void setFirst (Object first) { this.first=first; }
    Object getFirst () { return first; }
}

Pair p = new Pair();
p.setFirst (new Integer (5));
Integer i = (Integer) p.getFirst();  // narrowing
Why Generics?

Generic classes (universal polymorphism) is the perfect solution. The compiler checks that the class is used correctly.

```java
class Pair <T> {
    public final T first, second;
    Pair (T first, T second) {
        this.first=first; this.second=second;
    }
    T getFirst () { return first; }
}

Pair<Integer> p =
    new Pair<Integer>(new Integer(5), new Integer(8))
Integer i = p.getFirst();
Pair<String> q =
    new Pair<String>("abc","xyz");
String s = q.getFirst();
```
Wrapper Classes

Generics class can be instantiated only with classes (not primitive types).
However, Autoboxing and unboxing of primitive types make generics act as if they were applicable to primitive types. This is hugely useful.

```java
Pair<Integer> p = new Pair<>(5, 8);
Pair<Character> q = new Pair<>('a', 'b');
int i = p.getFirst();
char c = q.getFirst();
```
class Pair <T,U> {
    public final T first;
    public final U second;
    Pair (T first, U second) {
        this.first=first; this.second=second;
    }
    T getFirst () { return first; }
    U getSecond() { return second; }
}

Autoboxing and unboxing.

Pair<Integer,String> p = new Pair<>(5, "hello");
int i = p.getFirst();
String s = p.getSecond();
Generic method

A method can be generic even if the class it is in is not generic.

```java
public <T> T pick (T... choices) {
    if (choices.length == 0) return null;
    final int i = new Random().nextInt(choices.length);
    return choices[i];
}
```
Generic method

This does not work:

```java
public <T> T max (T t1, T t2) {
    if (t1 > t2) {
        return t1;
    } else {
        return t2;
    }
}
```
Generic method

This does not work:

```java
public <T> T max (T t1, T t2) {
    if (t1 > t2) {
        return t1;
    } else {
        return t2;
    }
}
```

```java
public <T extends Comparable<T>> T max (T t1, T t2) {
    if (t1.compareTo(t2) > 0) {
        return t1;
    } else {
        return t2;
    }
}
```
Three important Java generic interfaces:

1. Iterator
2. Comparable
3. Comparator

are often used in conjunction with Java collections.

Also important is:

1. AutoCloseable
Java 8 Functional Interfaces

**Table 2-1. Important functional interfaces in Java**

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Arguments</th>
<th>Returns</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate&lt;T&gt;</td>
<td>T</td>
<td>boolean</td>
<td>Has this album been released yet?</td>
</tr>
<tr>
<td>Consumer&lt;T&gt;</td>
<td>T</td>
<td>void</td>
<td>Printing out a value</td>
</tr>
<tr>
<td>Function&lt;T,R&gt;</td>
<td>T</td>
<td>R</td>
<td>Get the name from an Artist object</td>
</tr>
<tr>
<td>Supplier&lt;T&gt;</td>
<td>None</td>
<td>T</td>
<td>A factory method</td>
</tr>
<tr>
<td>UnaryOperator&lt;T&gt;</td>
<td>T</td>
<td>T</td>
<td>Logical not (!)</td>
</tr>
<tr>
<td>BinaryOperator&lt;T&gt;</td>
<td>(T, T)</td>
<td>T</td>
<td>Multiplying two numbers (*)</td>
</tr>
</tbody>
</table>
AutoCloseable

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource. An object that may hold resources (such as file or socket handles) until it is closed. The close() method of an AutoCloseable object is called automatically when exiting a try-with-resources block for which the object has been declared in the resource specification header. This construction ensures prompt release, avoiding resource exhaustion exceptions and errors that may otherwise occur.
Iterator

The interface `java.lang.Iterator<E>` has methods (i.a.):

- `boolean hasNext()`
- `E next()`

The interface `java.util.Collection` has method (i.a.):

- `Iterator<E> iterator()`

```java
public static void main (String[] args) {
    final List<Integer> list = new ArrayList<>();
    final Iterator<Integer> it = list.iterator();
    while (it.hasNext()) {
        // NB. auto unboxing, no narrowing
        final int x = it.next();
        // Do something with "x"
    }
}
```
I tend to use the for loop.

```java
public static void main (String[] args) {
    final List<Integer> list = new ArrayList<>();
    for (final Iterator<Integer> it = list.iterator();
         it.hasNext();
          //**/
    {
        final int x = it.next();
        // Do something with "x"
    }
}

(Interesting use of final in a for loop.)
```
The “for each” loop is better:

```java
public static void main (String[] args) {
    final List<Integer> list = new ArrayList<>();
    for (Integer x: list) {
        // Do something with "x"
    }
}
```

No narrowing necessary, and static type checking possible!
The interface `java.lang.Comparable<T>` has just one method.

```java
interface Comparable <T> {
    int compareTo (T other);
}
```

The interface is used to give a class a “natural” ordering — an ordering the Java API’s (especially the collection API’s) uses by default. It *should* be a total ordering consistent with equals: for every `e1` and `e2` of the class `T`, `e1.compareTo(e2)==0` iff `e1.equals(e2)`. 
// Give X a "natural" ordering

public class X implements Comparable<X> {
    /* ... */

    public int compareTo(X other) {
        /* compare 'this' and 'other' */

        if (/* 'this' is greater than 'other' */) {
            return +1;
        }

        if (/* 'this' is less than 'other' */) {
            return -1;
        }

        else {
            return 0;
        }
    }
}

// Give X a "natural" ordering

class X implements Comparable<X> {
    /* ... */
    int compareTo (X other) {
        /* compare 'this' and 'other' */
        if (/* 'this' is greater than 'other' */) {
            return +1;
        } else {
            /* Should be consistent with 'equals()' */
            return 0;
        }
    }
}
class Person implements Comparable<Person> {

    final String name;
    final int idNumber;

    /* Warning: This method is *not* consistent with equals; use for sorting only and not hashing. */
    @java.lang.Override
    int compareTo (final Person other) {
        if (this.idNumber > other.idNumber) {
            return +1;
        } else if (this.idNumber < other.idNumber) {
            return -1;
        } else {
            return this.name.compareTo (other.name);
        }
    }
}

For example:

```java
abstract static class Coin implements Comparable<Coin>
{
    abstract int value();
    public int compareTo(final Coin other) {
        final int v=value(), w=other.value();
        if (v<w) return -1;
        else if (v>w) return +1;
        else return 0;
    }
}
```
The interface java.util.Comparator<T> has two methods.

```java
interface Comparator <T> {
    int compare (T o1, T o2)
    boolean equals (Object obj);  // equal comparing
}
```

(We rarely create more than one instance of a particular comparator, so equals is hardly ever overridden.)

A comparator allows us to describe an ordering of objects without a natural ordering, or with a completely different ordering. This ordering is completely independent from the class.
Comparator

An example program which compares 2D points in two ways.
interface/Main.java
Comparator

The generic procedure `reserveOrder()` has some really advanced code ultimately relying on an unsafe cast to do the work. The code is something like this:

```java
class Collections {
    // ...

    private static final class RevComp<T> implements Comparator<T> {
        private static final RevComp<Object> INSTANCE = new RevComp<Object>();

        public int compare (Comparable<Object> o1, T o2) {
            Comparable<T> c2 = (Comparable<T>) o2;
            return c2.compare(o1);
        }
    }

    public static <T> Comparator<T> reverseOrder () {
        return (Comparator<T>) RevComp.INSTANCE;
    }
}
```
End of interfaces. Next Arrays and Collections classes??
The class `java.util.Arrays` has methods (i.a.):

```java
static void sort (int [] a)
static void sort (double [] a)
static void sort (Object [] a)
```

```java
class Main {
    public static void main (String [] args) {
        Coin [] xs;
        Arrays.sort (xs);
    }
}
```

[Notice that arrays are co-variant in Java so that this sort of call works. This leads to the problem of `ArrayStoreException`.]

Many predefined classes in Java are generic. Generics make these classes useful in more situations. The Java package `java.util` is filled with generic, “utility” classes — classes that are used to solve a wide variety of problems types.

Many of these classes are container or collection classes — classes that hold many individual sub-pieces (like arrays do). These classes are organized in a rich system known as the collection classes.

Collections classes share two important characteristics:

1. the individual items have arbitrary type
2. it is common to iterate over all the individual items of the collection
What is a list?
A list is an ordered collection of elements, like a chain.

There is a first element, a second element, and so on. There is a last element as well. Sometimes this structure is called a sequence. The same value may occur more than once (unlike a set).
List versus Array

The advantage of an array is that it is fairly cheap to allocate a big fixed amount of storage at once. But it wastes storage if a lot of requested, but little is needed. Also, removing an element in the middle of an array is expensive because one has to move a lot of elements.

A list allocates exactly what is needed as it is needed—no space is wasted. Removing an element in the middle can be cheaper than with an array. On the other hand, random access to all elements is lost and more bookkeeping is required. Access to the first and last elements of a list may be cheap.

ArrayList in an attempted compromise. How does it handle changes in the number of elements?
List

The Java API has already implemented the LinkedList class. Let us implement our own simple version.

Implementing our own lists:
list/IntList.java
list/GenericList.java

But first a mental picture of the approach:
Linked list represented by instances of two-part nodes
Linked List

Linked list with elements allocated in the heap
Linked List

Abstract representation of a linked list
Let us implement our own simple version of a linked list:
list/IntList.java
list/GenericList.java

The Java API has already implemented the LinkedList class, which like the ArrayList class, implements the generic List interface.
interface java.util.List<E> implements Collection<E>

E get (int index)
E set (int index, E element) returns previous element, if any
void add (int index, E element)
E remove (int index)

int size() number of elements in the list
void clear() removes all the elements

boolean contains (Object o) search
int indexOf (Object o) index of the first occurrence

Iterator<E> iterator ()
List Interface in the Collection Classes

The Java API has two implementations of the list interface with different performance characteristics.
ArrayList in the Collection Classes

java.util.ArrayList<E> implements List<E>

ArrayList()
ArrayList (int initialCapacity)

void ensureCapacity (int minCapacity)
void trimToSize ()
LinkedList in the Collection Classes

java.util.LinkedList<E> implements List<E>, Dequeue<E>

LinkedList()

E getFirst () same as element()
E getLast ()
void addFirst () same as push()
void addLast ()
E removeFirst () same as pop(), remove()
E removeLast ()
Lists

If you have to save space, then use a linked list. If you have to have fast access (set/get) to random elements, then use a list implemented as an array.
If you have to remove and add elements in the middle of the list, use a linked list. See the Josephus problem.
Collection Classes

Not be be confused with the generic interface `Collection<T>` is the utility class

```
java.util.Collections
```

```
static <T>  int    binarySearch(List<...> list, T key)
static <T>  List<T>  emptyList()
static <T>  List<T>  singletonList(T o)
static <T ...>  T   max (Collection<? extends T> coll)
static <T ...>  T    min (Collection<? extends T> coll)
static    int        frequency (Collection<?> coll, Object o)
static    void        shuffle (List<?> list)
static <T ...>  void  sort (List<T> list)
static <T>  List<T>  synchronizedList (List<T> list)
static <T>  List<T>  unmodifiableList (List<T> list)
```
Also in `java.util.Arrays`

```java
static <T> List<T> asList (T... a)
```

```java
List<Integer> list = Arrays.asList (1,2,3,4,5,6);
```

I never can remember where this very useful static method is found.
Avoid

Do not use `StringBuffer`, `Stack`, `Vector`, or `Hashtable`, unless you need synchronization.

Use `StringBuilder`, `ArrayDeque`, `ArrayList`, or `HashMap`, instead.
The class `java.util.Collections` has methods (i.a.):

```java
static void sort (List list);
static void sort (List list, Comparator c);
```

Notice the use of the interface `List`.

```java
class Main {
    // For some class X implementing Comparable<X>
    public static void main (String[] args) {
        final List<X> list = new ArrayList<X> ();
        add (new X());
        add (new X());
        Collections.sort (list);
        Collections.sort (list, Collections.reverseOrder ()�);
    }
}
```
Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names:

A **stack** is a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list.

A **queue** is a linear list for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end.

A **deque** ("double-ended queue") is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list. A deque is therefore more general than a stack or a queue; it has some properties in common with a deck of cards, and it is pronounced the same way. We also distinguish **output-restricted** or **input-restricted** deques, in which deletions or insertions, respectively, are allowed to take place at only one end.

From Knuth, volume I, page 235. The term deque was coined by E. J. Schweppe.
Stack, Queue, Deque

stack (LIFO): addFirst or push, removeFirst or pop
queue (FIFO): addLast (enqueue), removeFirst
deque: addFirst, addLast, removeFirst, removeLast
If you want a stack, a queue, or a deque in Java.

```java
Deque<Integer> stack = new ArrayDeque<Integer>();
Deque<String> queue = new ArrayDeque<String>();
Deque<Integer> deque = new ArrayDeque<Integer>();
```

The `LinkedList` class also implements the interface `Deque`. It is preferred only when it that data is treated as an ordered sequence in which it is necessary to insert elements.