## Java Generics

A Java variable can be one of the eight primitive types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Value Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>signed integers</td>
<td>0 to 127</td>
</tr>
<tr>
<td>short</td>
<td>signed integers</td>
<td>0 to 32767</td>
</tr>
<tr>
<td>int</td>
<td>signed integers</td>
<td>0 to 2147483647</td>
</tr>
<tr>
<td>long</td>
<td>signed integers</td>
<td>0 to 9223372036854775807</td>
</tr>
<tr>
<td>float</td>
<td>IEEE 754 floating point</td>
<td>-1.4E-45 to 3.4028235E+38</td>
</tr>
<tr>
<td>double</td>
<td>IEEE 754 floating point</td>
<td>-1.7976931348623157E+308</td>
</tr>
<tr>
<td>char</td>
<td>Unicode character</td>
<td>\u0000 to \uFFFF</td>
</tr>
<tr>
<td>boolean</td>
<td>true, false</td>
<td>false</td>
</tr>
</tbody>
</table>

Everything else is a reference to an object. An `Object` variable can hold a reference to any type of object.

### class Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone()</td>
<td>Creates and returns a copy of this object.</td>
</tr>
<tr>
<td>equals(Object obj)</td>
<td>Indicates whether some other object is &quot;equal to&quot; this one.</td>
</tr>
<tr>
<td>finalize()</td>
<td>Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.</td>
</tr>
<tr>
<td>getClass()</td>
<td>Returns the runtime class of an object.</td>
</tr>
<tr>
<td>hashCode()</td>
<td>Returns a hash code value for the object.</td>
</tr>
<tr>
<td>notify()</td>
<td>Wakes up a single thread that is waiting on this object's monitor.</td>
</tr>
<tr>
<td>notifyAll()</td>
<td>Wakes up all threads that are waiting on this object's monitor.</td>
</tr>
<tr>
<td>toString()</td>
<td>Returns a string representation of the object.</td>
</tr>
<tr>
<td>wait()</td>
<td>Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.</td>
</tr>
<tr>
<td>wait(long timeout)</td>
<td>Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.</td>
</tr>
<tr>
<td>wait(long timeout, int nanos)</td>
<td>Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.</td>
</tr>
</tbody>
</table>
Widening Conversion:

```java
String s = new String ("Foo");
Object obj;
obj = s;
```

Narrowing Conversion:

```java
String s = new String ("old String");
Object obj;
obj = s;
s = new String("new String");
s = (String) obj;
```

Java Collections hold Object references:

```java
ArrayList L = new ArrayList();
L.add(new Integer(1)); // Integer is a wrapper class
int x = (Integer) L.get(0);
L.add(2); // autobox : automatically convert from primitive to wrapper object Integer
int y = L.get(0); // auto-unbox
```

You can add any type of object to the list, but you have to specify what type of object you are taking out. What if there’s a mismatch?
Generics allow programmer to specify intent.

Without generics:

    List myIntList = new LinkedList();
    myIntList.add(new Integer(0));
    Integer x = (Integer) myIntList.iterator().next()

With generics:

    List<Integer> myIntList = new LinkedList<Integer>();
    myIntList.add(new Integer(0));
    Integer x = myIntList.iterator().next();

Excerpt from the definitions of interfaces List and Iterator in package java.util:

    public interface List<E> {
        void add(E x);
        Iterator<E> iterator();
    }

    public interface Iterator<E> {
        E next();
        boolean hasNext();
    }

A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration.

There aren’t multiple copies of the code: not in source, not in binary:

    List <String> l1 = new ArrayList<String>();
    List<Integer> l2 = new ArrayList<Integer>();
    System.out.println(l1.getClass() == l2.getClass());

Prints true, because all instances of a generic class have the same run-time class, regardless of their actual type parameters.
public class ArrayBag<E> implements Cloneable {
    private Object[] data;
    private int manyItems;
    public ArrayBag() {
        final int INITIAL_CAPACITY = 10;
        manyItems = 0;
        data = new Object[INITIAL_CAPACITY];
    }
    @SuppressWarnings("unchecked")
    // no semicolon
    public E grab() {
        ...
        return (E) data[i];
    }
    public void addAll(ArrayBag<E> addend) {
    }
    public int countOccurrences(E target) {
        ...
        if (target.equals(data[index]))
            answer++;
    }
    public boolean remove(E target) {
        // … find target at position index; overwrite target and set unused reference to null
        data[index] = data[manyItems];
        data[manyItems] = null;
        ...
    }
    @SuppressWarnings("unchecked")
    public void addMany(E... elements) {
        // variable Arity method : args put into an array
        if (manyItems + elements.length > data.length) {
            // Ensure twice as much space as needed.
            ensureCapacity((manyItems + elements.length)*2);
        }
        System.arraycopy(elements, 0, data, manyItems, elements.length);
        manyItems += elements.length;
    }
    // A generic method
    public static <E> ArrayBag<E> union(ArrayBag<E> b1, ArrayBag<E> b2) {
    }
}

Without the annotations: @SuppressWarnings("unchecked"): 

ArrayBag.java uses unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details.
A `Comparable` object implements the method `int compareTo(Object o)`, which also behaves like C's `strcmp`. `X.compareTo(y)`: return negative if x < y, 0 if x equals y, positive if x > y

```java
public class Person implements Comparable, Cloneable {
    private String name;   private int year;   private int month;  private int day;
    public Person(String name, int year, int month, int day) {
        this.name = new String(name); this.year = year;  this.month = month; this.day = day;
    }
    public String getName() { return name; }
    public Object clone() {
        try  {Person p = (Person) super.clone();  p.name = new String(name); return p;}
        catch (CloneNotSupportedException x) { throw new InternalError(x.toString()); }
    }
    public boolean equals(Object aThat) {
        if ( this == aThat ) return true;
        if ( !(aThat instanceof Person) ) return false;
        Person that = (Person)aThat;
        return name.equals(p.name)&& year == p.year&&month==p.month&&day == p.day;
    }
    public int hashCode() { // ensure different persons get different results
        int hval = name.hashCode()+year; hval = (hval << 4) + month; hval = (hval << 4) + day;
        return hval;
    }
    public String toString() {return '{' + name + ',
      month + '/' + day + '/' + year + '}';}
}
```

```java
public final class Person implements Comparable<Person> {
    public int compareTo(Person aThat) {     // Implement Comparable:
        if ( this == aThat ) return 0;
        Person p = (Person) o;
        int result = name.compareTo(p.name);
        return result;
    }
}
```

```java
class ArraysTest3 {
    static void search(Object[] a, Object n) {/* (same as in Figure 2)*/ }
    public static void main(String[] args) {
        // Build Array:
        Person[] array = new Person[3];
        array[0] = new Person("Horatio", 1835,12,6); array[1] =
            new Person("Charles",1897,3,11);
        array[2] = new Person("Albert",1901,1,20);
        Arrays.sort(array);
    }
}
```

```java
public final class Person implements Comparable<Person> {
    public int compareTo(Person aThat) {     // Implement Comparable:
        if ( this == aThat ) return 0;
        int result = name.compareTo(p.name);
        return result;
    }
}
Or search on name using a **Comparator**:

```java
class ByName implements Comparator {
    public int compare(Object o1, Object o2) {
        Person p1 = (Person) o1; Person p2 = (Person) o2;
        return p1.getName().compareTo(p2.getName());
    }
}
```

```java
class ArraysTest4 {
    static ByName comp = new ByName();
    public static void main(String[] args) {
        // Build Array:
        Person[] array = new Person[3];
        array[0] = new Person("Horatio", 1835,12,6);
        array[1] = new Person("Charles",1897,3,11);
        array[2] = new Person("Albert",1901,1,20);
        Arrays.sort(array, comp);
    }
}
```
Legal?
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;

What if we execute:
   lo.add(new Object());
   String s = lo.get(0); // attempts to assign an Object to a String

“lo = ls;” will cause a compilation error

Rule: In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is not the case that G<Foo> is a subtype of G<Bar>.

How to deal with the fact that we’re not sure what’s in the collection?

Wildcards

    void printCollection(Collection c) {
        Iterator i = c.iterator();
        for (k = 0; k < c.size(); k++) {
            System.out.println(i.next()); // okay, object is a supertype of all objects
        }
    }

    void printCollection(Collection<Object> c) { // generic,
        for (Object e : c) {
            System.out.println(e);
        }
    }

Problem: Collection<object> is not a supertype of all collections.

Q: So what is the supertype of all kinds of collections?
A: Collection<?> // “collection of unknown” whose element type matches anything.
```java
void printCollection(Collection<?> c) {  // “wildcard type”
    for (Object e : c) {   // elements are, in fact, Objects
        System.out.println(e);
    }
}
```

But this is an error:
```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // compile time error
```

Any parameter we pass to `add()` would have to be a subtype of the unknown type.

Since we don’t know what type that is, we cannot pass anything in (except null, which is a member of every type).

Q: You can’t add anything to `Collection<?>`, so what is it useful for?
A: You can read its contents, as shown above.
public abstract class Shape {
    public abstract void draw(Canvas c);
}
public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { ... }
}
public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { ... }
}

These classes can be drawn on a canvas:

public class Canvas {
    public void draw(Shape s) {
        s.draw(this);
    }
}

A method in Canvas that draws them all:

    public void drawAll(List<Shape> shapes) {
        for (Shape s: shapes) {
            s.draw(this);
        }
    }

But: drawAll() can only be called on lists of exactly Shape.
A method that accepts a list of any kind of Shape:

```java
public void drawAll(List<? extends Shape> shapes) { ... }
```

Unknown type is a subtype of Shape: Shape is the “upper bound” of the wildcard. We can call drawall() on List<Circle> if we want.

But it is illegal to add to shapes in the body of the method:

```java
public void addRectangle(List<? extends Shape> shapes) {
    shapes.add(0, new Rectangle()); // compile-time error!
}
```

The type of the second parameter to shapes.add() must be “? extends Shape” - an unknown subtype of Shape.

Since we don’t know what type it is, we don’t know if it is a supertype of Rectangle so it isn’t safe to pass a Rectangle there.
Generic Methods

Write a method that takes an array of objects and a collection and puts all objects in the array into the collection.

Wrong:

```java
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // compile time error: can’t put objects into a collection of unknown type
    }
}
```

```java
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // correct
    }
}
```

Element type of collection must be a super-type of the element type of the array (due to the cast to T of each element of a.)

The type parameter T is used twice, in order to express a dependency between the type of the collection elements and the type of the array elements.

```java
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferred by compiler to be Object
```

```java
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferred to be String
```

```java
fromArrayToCollection(sa, co);// T inferred to be Object
```

```java
Number[] na = new Number[100];
fromArrayToCollection(na, cs);// compile-time error
```
Should I use generic methods or wildcard types?

    interface Collection<E> {
        public boolean containsAll(Collection<?> c);
        public boolean addAll(Collection<? extends E> c);
    }

vs.

    interface Collection<E> {
        public <T> boolean containsAll(Collection<T> c);
        public <T extends E> boolean addAll(Collection<T> c);
    }

Note: T is used only once, for polymorphism.

Its only effect is to allow a variety of actual argument types to be used at different invocation sites ⇒ use wildcards to support flexible subtyping.

We can also use both generic methods and wildcards in tandem.

    class Collections {
        public static <T> void copy(List<T> dest, List<? extends T> src){...}
    }

Wildcard for subtyping: the element type of src can be any subtype of T.

Type parameter to express dependency: Any object copied from the source list src must be assignable to the element type T of the destination list dst.