Generics

Polymorphism allows us to write general code that works for a variety of types that are related and have the same methods (even if different implementations. Sometimes code is so general it should work for any type at all. For this we use generics.

Here is a case where we have the same code for several methods, with the only difference being the type of the array we pass in.

```
 public static Animal getLast(Animal[] list) {
      if (list == null) return null;
      return list[list.length - 1];
 }

 public static Worker getLast(Worker[] list) {
      if (list == null) return null;
      return list[list.length - 1];
 }

 public static GameItem getLast(GameItem[] list) {
      if (list == null) return null;
      return list[list.length - 1];
 }

//… lost will to code
```

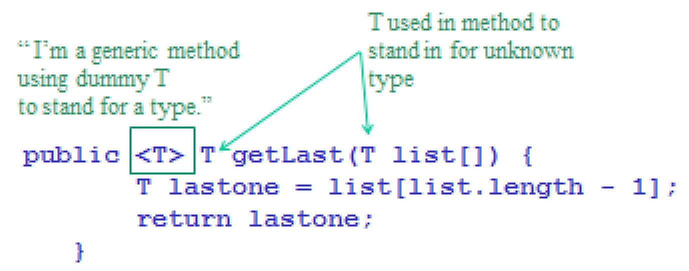
It almost certainly isn't appropriate to put all of these in the same inheritance hierarchy when they're not meaningfully related, just to get around having lots of methods. Of course, we could always use the fact that everything inherits from Object...

```
  public Object getLast(Object[] list) {
       if (list == null) return null;
       return list[list.length - 1];
  }


 public otherMethod() {
       Cat [ ] catlist = {new Cat("Muffy"),
                          new Cat("Fluffy")};

       Cat c = getLast(catlist); // getLast returned Object
       Cat c = (Cat)(getLast(catlist));
```

This gets us out of writing multiple versions of the method, but now Java can't predict what object type we'll actually get out of the method, since it just says it returns Object. So we will have to cast it. This is clumsy and annoying.

Generic Methods

A generic method uses a dummy to stand in for a type to be specified later. This type parameter is enclosed in angle brackets, and by convention we use a single uppercase letter for it. Throughout the method, we use the type parameter as if it were a real known type.


"I'm a generic method using dummy T to stand for a type."
T used in method to stand in for unknown type

```
 public <T> T getLast(T list[]) {
        T lastone = list[list.length - 1];
        return lastone;
    }
public otherMethod() {
      Cat [ ] catlist = {new Cat("Muffy"),
                         new Cat("Fluffy")};
      Cat c = getLast(catlist); // no cast, T = Cat

      Worker[] team = {new Bee(), new Robot(), new Robot()}
      Worker w = getLast(team); // no cast, T = Worker
```

Based on the type passed to the method, the compiler can determine what real type the type parameter stands for in any particular case, and can therefore check the type being returned, so we no longer have to cast.

We can have more than one type parameter, if we need to handle more than one type in the same method.

```
public <T, R> void showTogether(T[] tlist, R[] rlist ) {
      int minlen = tlist.length < rlist.length?
                   tlist.length : rlist.length;

      for (int i = 0; i < minlen; i++) {
            System.out.println(tlist[i] + " - "+ rlist[i]);
        }
}

public otherMethod() {
      Cat [ ] catlist = {new Cat("Muffy"),
                         new Cat("Fluffy")};

      Worker[] team = {new Bee(), new Robot(), new Robot()}

      showTogether(catlist, team); // T=Cat, R=Worker
      showTogether(team, catlist); // T=Worker, R=Cat
}
```

Type Erasure

In fact, behind the scenes, when Java is compiling generic code, it turns it into Object polymorphism and inserts all casts necessary to keep the code type-safe. Or, if the types are bounded (see later) it uses the bounding type for polymorphism.

Autoboxing

Since under the hood, generics rely on Object polymorphism, generics won't work with primitive types such as int and boolean. However, Java has wrapper classes defined for each primitive type, and can generally automatically wrap a primitive in an object of the related type, called autoboxing and automatically unbox as well.

```
 public <T> T getLast(T[] list) {
      if (list == null) return null;
      return list[list.length - 1];
 }

public otherMethod() {
      Integer[] list = {5, 10, 15, 20};
      Integer ii = getLast(list);
      int i = getLast(list); // automagic cast to int
}
```

Generics and new

Java does not allow using new with a generic type parameter. This makes sense. We can't say

```
T var = new T();
```

because we can't be sure every type T stands for will have a default constructor, and type erasure would mean we would actually be making an object of type Object anyway.

For similar reasons, we cannot create an array of a generic type. However we can create an array of Objects and cast it to the generic type.

```
public static <T> T[] startArray(T firstitem, int len) {
      //T[] tlist = new T[len]; // java doesn't allow

      // workaround
      T[] tlist = (T[])(new Object[len]);

      // now can treat it as an array of T's.
      tlist[0] = firstitem;

      return tlist;
}
```

In general, if we need an array-like structure for generic code, instead we use one that is already generic itself, an ArrayList.

ArrayList

ArrayList is a built in Java class declared with type parameters – a generic class. When we declare an ArrayList variable, it has parameterized type – it's type is ArrayList<T> for some type T. Java will let you get away with not specifying the type parameter, but this will make your code less type safe[1].

The main drawback of arrays (other than the difficulty in creating them at all in generic code) is that they are not dynamic data structures. You must choose a length for the array object when you create it, and then if you need to change its size later, you must create a new object and copy the contents over. Also there is no way to insert new array elements between existing ones or delete an element without extra blank elements between.

ArrayLists automatically resize themselves as you add and remove elements. The add method allows you to add methods, and puts them into the structure in the order added.

```
// it will grow as needed
ArrayList<String> rainbow = new ArrayList<String>();

rainbow.add("red");
rainbow.add("yellow");
rainbow.add("green");
rainbow.add("blue");
rainbow.add("purple");

// insert after red
rainbow.add(rainbow.indexOf("red") + 1, "orange");
```

The elements in the ArrayList are numbered with indices like the elements in an array (starting from 0). The get method takes an index and returns the element at that index. Given an element that is in the ArrayList, the indexOf method returns its index.

The add method is also overloaded to allow adding at a particular index, which instead of replacing the existing item, moves it and all following elements down by one. The remove method removes an element and moves all the following elements up by one.

ArrayList has many other useful methods you can look up in the Java API
http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Iterators

One drawback of using an ArrayList over an array is that we no longer have direct access to the structure, and have to go through methods. Using a regular for loop to go through an ArrayList is a little clumsy; however ArrayList does work nicely with a for each loop. What the for each loop is actually doing, under the hood, is using a separate object called an iterator, which knows how to move through the ArrayList.

Iterators typically have methods

[1] And make other programmers suspect you don't actually understand generics.

- hasNext – returns a boolean saying whether there is at least one more element in the list after the iterator's current position
- nextIndex—returns in the index of the element after the iterator's current position
- next—returns the next element in the list and updates the iterator's current position to after this item

as well as equivalent methods for moving backwards, hasPrevious, previousIndex, previous.

```
ArrayList<Cat> cattery = new ArrayList<Cat>();

cattery.add(new Cat("Muffy", 2));
cattery.add(new Cat("Fluffy", 5));
cattery.add(new Cat("Clyde", 3));

// object that knows how to go through elements
ListIterator caterator = cattery.listIterator();

// true if there is a next element (still at same position)
while(caterator.hasNext()){
   // .next() is the next element (position has changed)
   // generic, so next is whatever type is in list
   caterator.next().miaow();
}

// now caterator at end of list, no more next
```

This allows us to move forward and backward through an ArrayList. It also means that we can keep our place in an ArrayList long-term. For instance, if we have an instance variable in a class whose type is an iterator for some ArrayList, the class can have methods that process whatever the current element of the ArrayList is, and keep track of what the next one should be the next time they are called.

Note that in general the behavior of an iterator if the underlying data structure is changed is not guaranteed.

Collections

In addition to ArrayList, java has many built-in generic data structures for different needs.

LinkedList is similar to ArrayList, but does not maintain indices and instead makes removing or adding elements in the middle of the structure much more efficient. A Set ensures that there are no duplicate items, and has no particular order.

Stack and PriorityQueue represent two common data structures. In a stack, elements are typically added and removed on the same end, so the newest element is the next one to be taken off. In a queue, elements are typically added and removed on opposite ends, so the oldest element is the next one to be taken off. (A PriorityQueue is someone more complex, allowing elements to jump the queue and be dealt with sooner depending on their priority.)

A Map allows us to represent a mapping (association) between keys and values.

Generic Classes

We often use built-in generic classes, particularly from the Collection hierarchy, but we can also define our own, if we have a structure that is always the same, but may contain different types. In this case we include the type parameters immediately after the name of the class, and can then use these types anywhere in the class.

```
public class MyPair <A, B> {
    private A first;
    private B second;

    public MyPair(A vala, B valb){
      first = vala;
      second = valb;
    }

    public A getFirst() {
      return first;
    }

    public void setFirst(A val) {
      first = val;
    }

    // ...
```

Variables based on a generic class should be declared with a parameterized type.

```
MyPair<Cat, String> p =
  new MyPair<Cat, String>(new Cat("Fluffy"), "Cat");

// okay since we've given type
p.getFirst().miaow();

MyPair<String, Cat> p2 =
  new MyPair<String, Cat>("Other Cat", new Cat("Muffy"));

 p2.getSecond().miaow();
```

Note that again, Java takes care of keeping track of what types the type parameters stand for in any particular case, so we don't have to worry about casting.

Generic classes can inherit from non-generic classes or from other generic classes, and vice versa, e.g.:

```
class MyPair <A, B> extends Namey{
class MyTriple <A, B, C> extends MyPair<A,B>{
```

```
class TwoStrs extends MyPair <String, String> {
```

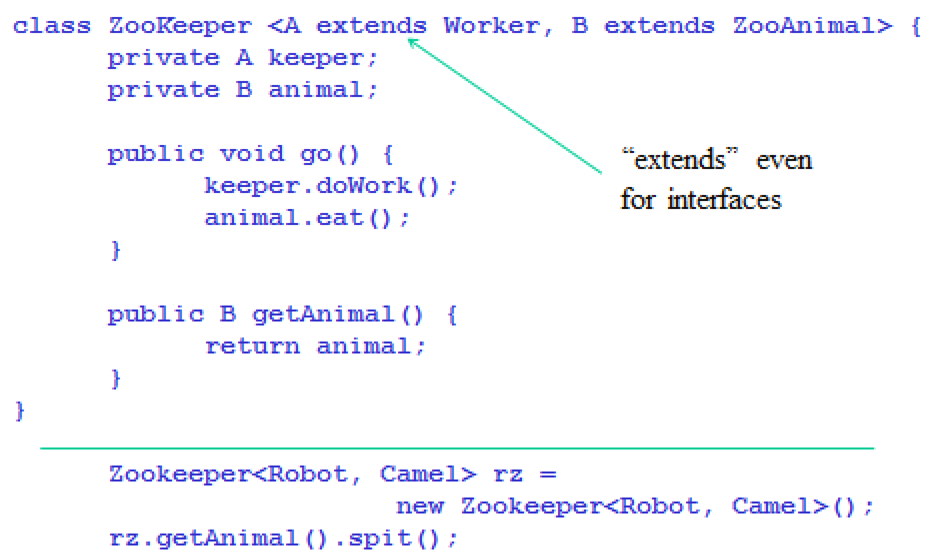
This says that a MyPair has everything a Namey (assumed to be a non-generic class) has, and can also use the type parameters A and B. MyTriple adds a third type parameter. TwoStrs makes MyPair specifically use String for both A and B.

Bounding Generics

For classes or methods, we sometimes want to bound our type parameters for generic code, that is, to say that the generic code applies only to classes within a certain inheritance tree or implementing a certain interface. In this case we use extends after the type parameter.

<A extends AClass> means that A can be any type that an AClass variable could polymorphically point to.

For bounded types, we can use methods that are defined for the bounding type.



```
class ZooKeeper <A extends Worker, B extends ZooAnimal> {
      private A keeper;
      private B animal;

      public void go() {
            keeper.doWork();
            animal.eat();
      }

      public B getAnimal() {
            return animal;
      }
}
```

```
      Zookeeper<Robot, Camel> rz =
                        new Zookeeper<Robot, Camel>();
      rz.getAnimal().spit();
```

So when do we want polymorphism and when do we want generics with bounded types?

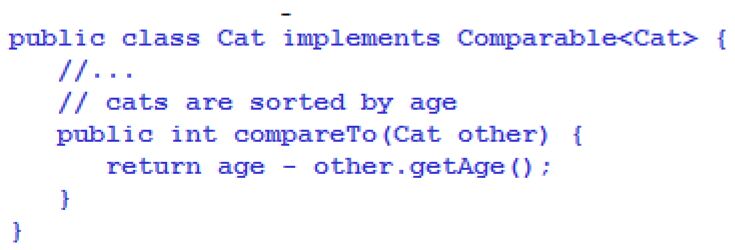
Again, type erasure means that the compiled version will be polymorphism anyway. So we should use generics when we want, in addition to what we get from polymorphism, for Java to do the type checking for us so that we can use the actual type without worrying about checking instanceof and casting. For instance, in the above, Java guarantees for us that if we have a Zookeeper<Robot, Camel> then the animal type we get out will be Camel, and will have Camel-specific methods, while if we have Zookeeper<Robot, Sloth> the animal type will be a Sloth, with any Sloth-specific methods.

Comparable and Comparator

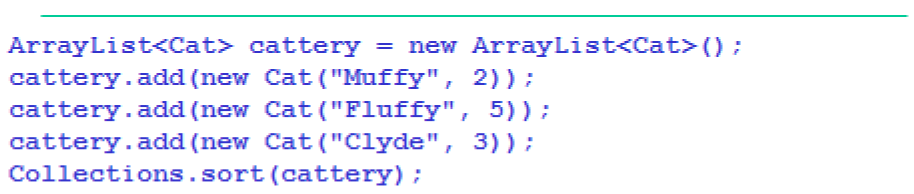
The Collections class provides a method sort which will sort the contents of an ArrayList. But it can only sort them if there is some kind of ordering defined for the type, that is, if we provide a way of knowing if one instance is less than another.

To do this, we made the class implement the generic interface Comparable<T> where T is the class itself.

To implement this interface, it must have a method CompareTo which takes a parameter of its own type and returns an int. The standard is that compareTo will return 0 if this is equal to the one being passed in, a positive number if this is greater, and a negative number if the other is greater. In this example, Cats are to be sorted by increasing age.



```
public class Cat implements Comparable<Cat> {
   //...
   // cats are sorted by age
   public int compareTo(Cat other) {
      return age - other.getAge();
   }
}
```



```
ArrayList<Cat> cattery = new ArrayList<Cat>();
cattery.add(new Cat("Muffy", 2));
cattery.add(new Cat("Fluffy", 5));
cattery.add(new Cat("Clyde", 3));
Collections.sort(cattery);
```

Some compareTo methods simply return 0, 1, or -1, while for others the size of the number returned reflects how much less or more this was than the one passed in.

If it makes sense to sort instance of a class, implementing Comparable<T> and putting the appropriate comparison for the sort in the class is the right way to handle it.

There are many cases, however, where we may want to sort items by different parameters in different circumstances. In that case, the compareTo method should be whatever we consider the default, and we can add other strategies for other comparisons by creating classes that implement Comparator<T> where T is our class.

Comparator<T> requires a method compare that returns an int using the same scheme as compareTo, but takes two parameters of the type. Since compareTo is inside the class, it is from the POV of an instance, and we only need to pass in one other. The compare method is generally in a separate class, so it needs two to be passed in.

```
public class CompCatByName implements Comparator<Cat> {
   public int compare (Cat c1, Cat c2) {
      int compname = c1.getName().compareTo(c2.getName());
      if (compname != 0) { // names different
            return compname;
      }
      return c1.compareTo(c2);// same name
   }
}
```

```
ArrayList<Cat> cattery = new ArrayList<Cat>();
cattery.add(new Cat("Muffy", 2));
cattery.add(new Cat("Fluffy", 5));
cattery.add(new Cat("Clyde", 3));
Collections.sort(cattery , new CompCatByName());
```

In this example, this secondary comparison for Cats first tries to order them alphabetically (taking advantage of String's compareTo method). If this comes out nonzero, we're done. If the cats have the same name, however, it falls back on ordering them by age.

Note that if we want to use this alternate comparison when we sort, we need to pass an instance of the strategy class into the sort method.