Prepared by Linan Qiu <lq2137@columbia.edu>

Generics

Life Without Generics

Generics. Is. A. Lie. Repeat that at least 5 times before you proceed to the next paragraph.

They sound scary, but they're nothing of that sort. Let's see why they exist.

Let's say you want to create a class that holds on to a certain primitive data type, say an `int`. You can do this in a class called `BoxInteger`

```
// BoxInteger.java

public class BoxInteger {
  private int data;

  public BoxInteger(int data) {
    this.data = data;
  }

  public int getData() {
    return data;
  }

  public static void main(String[] args) {
    BoxInteger myInteger = new BoxInteger(259);
    System.out.println(myInteger.getData());
  }
}
```

Let's say you want to use the Box for a `String` instead of an int. What do you do? You literally `ctrl-f` / `cmd-f` for the word "int" and replace it with "String". Okay not that literally, since Intege still has to be changed to String but you get the gist.

```
// BoxString.java

public class BoxString {
  private String data;

  public BoxString(String data) {
```

```
    this.data = data;
  }

  public String getData() {
    return data;
  }

  public static void main(String[] args) {
    BoxString myString = new BoxString("Han Solo dies");
    System.out.println(myString.getData());
  }
}
```

What if you want to do a `BoxDouble`? Or a `BoxChar`? Or a `BoxPerson` for a `Person` class you may have? This gets pretty tiring and your folder of source code goes crazy with tons of `BoxXXX.java` files. **This is not being lazy. This is not good.** You, being lazy you, ask the Java gods if there's a better solution.

Generics To The Rescue

Generics solve this precise problem. Now we create a `BoxGeneric` like this:

```
// BoxGeneric.java

public class BoxGeneric<SomeClass> {
  private SomeClass data;

  public BoxGeneric(SomeClass data) {
    this.data = data;
  }

  public SomeClass getData() {
    return data;
  }

  public static void main(String[] args) {
    BoxGeneric<Integer> myBox = new BoxGeneric<Integer>(259);
    System.out.println(myBox.getData());
    BoxGeneric<String> myBox2 = new BoxGeneric<String>("cow");
    System.out.println(myBox2.getData());
  }
}
```

What happens when the compiler compiles this code is that it realizes you want to use `BoxGeneric<Integer>` and `BoxGeneric<String>`, ie. use `BoxGeneric` on

a `Integer` and a `String`. It replaces all of `SomeClass` with `Integer` for `myBox` during compilation, and replaces all of `SomeClass` with `String` for `myBox2` during compilation as well. Then you have effectively two different classes all specialized for the specific data that you want to contain.

You've effectively created two classes (`BoxGeneric<Integer>` and `BoxGeneric<String>`) from a "template" version (`BoxGeneric`). Hence, generics are just a way for the compiler to create specialized version of classes on the fly instead of you having to write all of them. It's like a template or a generator that goes "fill in the blanks!" That's why it's a lie: it's just a very very smart find and replace.

The `SomeClass` in the code is the placeholder. It can be anything: `AnyClass`, `AnyType`, `Any` etc. Usually, for brevity, we use `T`. Hence, a presentable version of this code would look like this:

```
// BoxGeneric.java

public class BoxGeneric<T> {
  private T data;

  public BoxGeneric(T data) {
    this.data = data;
  }

  public T getData() {
    return data;
  }

  public static void main(String[] args) {
    BoxGeneric<Integer> myBox = new BoxGeneric<Integer>(259);
    System.out.println(myBox.getData());
    BoxGeneric<String> myBox2 = new BoxGeneric<String>("cow");
    System.out.println(myBox2.getData());
  }
}
```

You Probably Already Used Generics

You've probably used generics before without knowing it. Have you used `ArrayList`? Then you've used generics. Think about it: `ArrayList` is essentially stuck in the same situation as our `Box`. It should be able to fit every class possible, yet Java doesn't want to write `ArrayListInteger` and `ArrayListString` and all the other classes possible. So Java made `ArrayList` a generic. What else did you think you were doing when you write `ArrayList<String> list = new ArrayList<String>()`?

Declaring Generics

Before Java 7, we initialize generic classes as such:

```
BoxGeneric<Integer> myBox = new BoxGeneric<Integer>();
ArrayList<Double> list = new ArrayList<Double>();
```

Starting with Java 7, the guys at Oracle realized that the second `BoxGeneric<Integer>` or `ArrayList<Double>` is really annoying, so they shortened this to:

```
BoxGeneric<Integer> myBox = new BoxGeneric<>();
ArrayList<Double> list = new ArrayList<>();
```

Using Generics

You can only use generics with objects. **You cannot use them with primitives.** You'd get a compilation error. If you want to use `int` or `double` or other primitives, you have to use the **boxed** version of them: `Integer`, `Double` etc. This is because primitives are not objects.

In Java, all classes inherit from the `Object` class. Any class you create will automatically `extend Object`. The inner mechanism of generics require that the substitution be done only on `Object`s, and primitives aren't `Object`s.