Java Fundamentals 2 - Day 3

Generics

- Let’s put some apples and pears in boxes!

The Apple Box

- Suppose you have box, to contain an apple in:

```
public class Box {

  private Apple apple;

  public Apple get() {
    return apple;
  }

  public void set(Apple apple) {
    this.apple = apple;
  }
}
```

- What happens if you now get a pear? Does it fit in the box?

The Pear Box

- To fit a Pear, you need a new box:

```
public class PearBox {

  private Pear pear;

  public Pear get() {
    return pear;
  }

  public void set(Pear pear) {
    this.pear = pear;
  }
}
```

- The box is the same, but it needs a new name
- Now what if we get an orage?
- Is there no better box?

The Fruit Box

- Let’s make an interface `Fruit` and see if it helps:

```
interface Fruit {}
class Apple implements Fruit {}
class Pear implements Fruit {}

public class FruitBox {

  private Fruit fruit;

  public Fruit get() {
```

```
    return fruit;
  }

  public void set(Fruit fruit) {
    this.fruit = fruit;
  }
}
```

The Fruit Box 2

- Is this better?
- Are there any annoyances still?

The Annoying Fruit Box

- What happens if we use the `FruitBox`:

```
Apple myApple = new Apple();
FruitBox myBox = new FruitBox();
myBox.set(myApple);

// We lose type information on the Apple here!
Fruit myFruit = myBox.get();
```

- Can we do even better?

The Better Box

- Yes we can! With generics!
- Generics allow us to use a *type* as a parameter:

```
public class GenericBox<T> {

  private T item;

  public T get() {
    return item;
  }

  public void set(T newItem) {
    item = newItem;
  }
}
```

- The `T` is a parameter where we can put a *type* in
    - A class
    - An interface

The Better Box in Use

- To use the `GenericBox` we can do this:

```
Apple myApple = new Apple();
GenericBox myAppleBox = new GenericBox<Apple>();
```

```
myAppleBox.set(myApple);

// We still get an Apple here!
Apple myFruit = myAppleBox.get();

// And for Pears:
Pear myPear = new Pear();
GenericBox myPearBox = new GenericBox<Pear>();
myPearBox.set(myPear);

// We still get a Pear here!
Pear myOtherFruit= myPearBox.get();
```

Generics

- Generics allow us to add a *type* parameter to
    - Classes
    - Interfaces
    - Methods
- This allows the use of strong typing without losing information
- Fewer casts, more shared code

Bounding Generics

- What if we had a `Box` that was only intended for `Fruit`?

```
public class FruitBox<T extends Fruit> {
```

```
  private T item;

  public T get() {
    return item;
  }

  public void set(T newItem) {
    item = newItem;
  }
}
```

- The `FruitBox` is still generic, but only allows classes that implement `Fruit` as its parameter
- This allows you to put fruit-specific behaviour in the box and enfore only `Fruit` can be put inside

Generics in the JDK

- Java Collections use generics to make lists, maps and sets easy to use

```
List<String> myStrings = new ArrayList<String> ();

Map<Integer, User> usersById = new HashMap<Integer, User> ();

Set<User> allUsersOrderingBread = new HashSet<User> ();
```

- You will also find generics in other places:

    - `Comparator<T>`
    - `Callable<V>`

    - `Function<T,R>`

- Type parameter usually refers to first letter of something:
    - `T` = Type (very generic)
    - `V` = Value (mostly to pass a value to a function)
    - `R` = Return (for the type of a return value of a function)

Crazy Generics

- Consider creating a map between a `userId` and the combination of the `User` and all of their `Order`s:

```
Map<Integer, Pair<User, List<Order>>> usersByIdWithOrders = new HashMap<Integer, Pair<User, List<Order>>>();
```

- All that typing and bracket-matching! OMG!
- Thankfully, we have the diamond operator (`<>`):

```
Map<Integer, Pair<User, List<Order>>> usersByIdWithOrders = new HashMap<> ();
```

- The compiler fills in the same generics as the left side of the `=`

Generics in other places

- Two more places where you may see generics:
    - Interfaces

    - Methods
- These are important to have seen, to understand the "flow of information"

Generics in interfaces

- When an interface has a type parameter, we have a choice at implementation:
    - Choose a type parameter

```
public class MyStringComparator implements Comparator<String> {
  // ...
}
```

- Pass on the generics

```
public class MyGenericComparator<T> implements Comparator<T> {
  // ...
}
```

Generics in methods

- Methods can specify they work on multiple types
    - Usually it is inferred and not explicitly noted
    - This is called "capturing" a type

```
public <T> T doSomethingWithIt(T theThing) { ... }
```

```
String test = "Hello!";
String result = doSomethingWithIt(test);

Integer myNumber = 2873;
Integer secondResult = doSomethingWithIt(myNumber);
```

Other Generic Things

- Generics can be made really complex
    - They support upper bounds (`T extends Fruit`)
    - They support lower bounds (`T super Apple`)
    - They can be combined (`T extends Fruit & super Elstar`)
    - They support wildcards (`List<?>`)
- Remember PECS: "Producer Extends, Consumer Super"
- Rule of thumb: if your generics include upper and/or lower bounds, you are probably making them *too* complicated. Rethink your strategy!