Java Generics cheat sheet

Basics

Generics don't exist at runtime!

```
class Pair<T1, T2> { /* ... */ }
```
-- the type parameter section, in angle brackets, specifies type variables.

Type parameters are substituted when objects are instantiated.

```
Pair<String, Long> p1 = new
  Pair<String, Long> ("RL", 43L);
```

Avoid verbosity with the diamond operator:

```
Pair<String, Long> p1 =
  new Pair<>("RL", 43L);
```

Wildcards

`Collection<Object>` - heterogenous, any object goes in.
`Collection<?>` - homogenous collection of arbitrary type.

Avoid using wildcards in return types!

Intersection types

```
<T extends Object &
  Comparable<? super T>> T
max(Collection<? extends T> coll)
```

The return type here is ***Object***!

Compiler generates the bytecode for the most general method only.

Producer Extends Consumer Super (PECS)

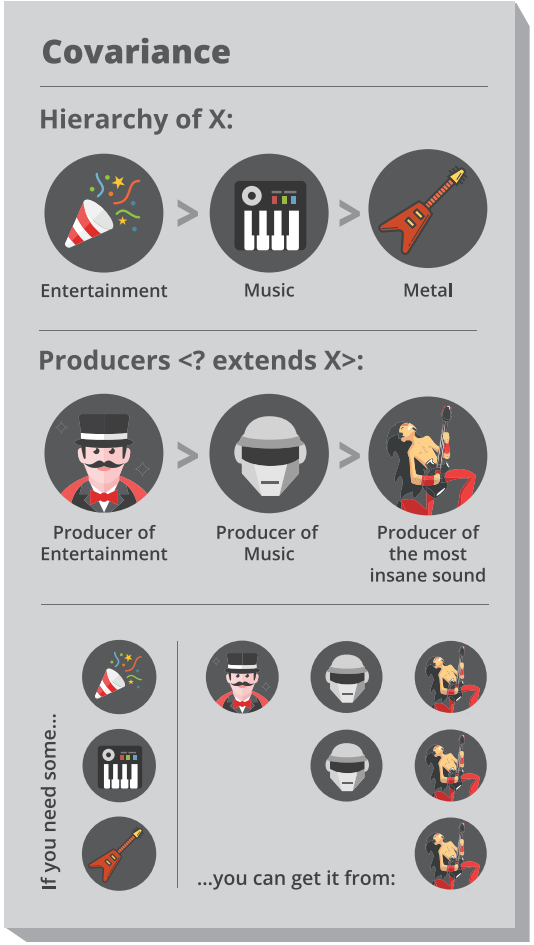
```
Collections.copy(List<? super T> dest, List<? extends T> src)
```

src -- contains elements of type T or its subtypes.

dest -- accepts elements, so defined to use T or its supertypes.

*Consumers are **contravariant** (use super).*

*Producers are **covariant** (use extends).*



Method Overloading

```
String f(Object s) {
  return "object";
}
String f(String s) {
  return "string";
}
<T> String generic(T t) {
  return f(t);
}
```

If called **generic("string")** returns "object".

Recursive generics

Recursive generics add constraints to your type variables. This helps the compiler to better understand your types and API.

```
interface Cloneable<T extends
Cloneable<T>> {
  T clone();
}
```

Now `cloneable.clone().clone()` will compile.

Covariance

```
List<Number> ⊁ ArrayList<Integer>
```

Collections are not covariant!