# Kotlin Generics

# Generics Classes

Classes in Kotlin can have type parameters.

These classes are called *generic types*.

Concrete types are created by instantiating the type parameters of a generic type.

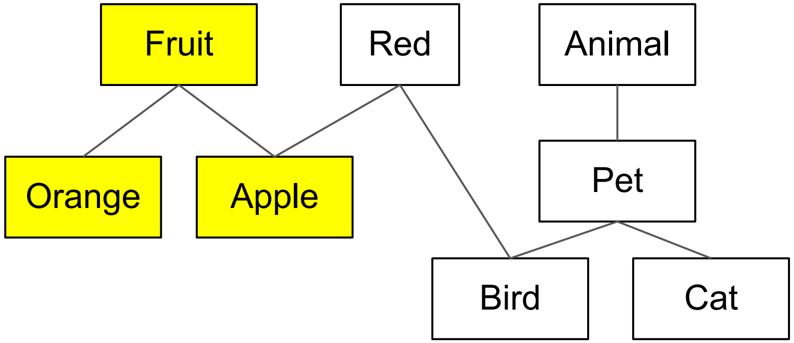
Kotlin can perform type inferences as long as sufficient information is available.

```
class Box<T>(val value: T) {
    fun getValue(): T = this.value
}


// specify all the types
val box: Box<Int> = Box<Int>(1)

// Omit L-value type
val box = Box<Int>(1)

// Omit type parameter
val box = Box(1)
```

# Constrained Generics

What if we want to describe a situation where the box must contain a fruit?

- Box<T>
- T needs to have an upper bound
  - <T : U>
  - where T: U

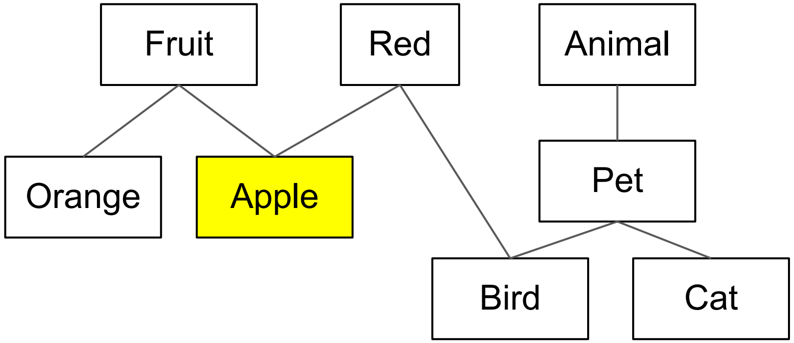



```
// Constraint inside <>

class Box<T:Fruit>(
  val value: T
)

// Explicit where-clause

class Box<T>(
  val value: T
) where T:Fruit

val x = Box(apple)   ✅
val y = Box(orange)  ✅
val z = Box(cat)     ❌
```

# Constrained Generics

What if we want a box to contain *red fruits*?

- Multiple constraints on <T>
- Must use where-clause




```
// Explicit where-clause

class Box<T>(
  val value: T
)
where T:Fruit,
      T:Red {
  ...
}

val x = Box(apple)   ✅
val y = Box(orange)  ❌
val z = Box(cat)     ❌
```

# Kotlin built-in collections

Kotlin provides built-in implementations of:

1. List
2. Set
3. Map

Each collection type comes with:

- Functional version (readonly)
- Mutable version

```
// a read-only list
List<T>

// a mutable list
MutableList<T>

// a read-only set
Set<T>

// a mutable set
MutableSet<T>

// a read-only map
Map<S,T>

// a mutable map
MutableMap<S,T>
```

# Generic Functions

How do we define a function that always returns the *first* element of a list?

```
List<T> -> T?
```

- Why is the result type T?
- Kotlin functions can have type parameters.

```
// Generic function

fun <T> first(xs: List<T>): T? {
  if(xs.isEmpty()) {
    return null
  } else {
    return xs.get(0)
  }
}

// Constraints on type parameters

fun <T:U> f(...) {
  ...
}

fun <T> f(...)
    where T:U,
          T:V {
  ...
}
```

# Case Study: Kotlin List

Kotlin provides built-in implementations of:

1. List
2. Set
3. Map

Each collection type comes with:

- Functional version (readonly)
- Mutable version

```
// Built-in list constructor:

fun <T> listOf(vararg xs:T): List<T>
fun <T> mutableListOf(vararg xs:T): MutableList<T>

// Example
val names = mutableListOf("Jack", "Jill")
names.add("Joe")

names += "Jason"
names += listOf("Jennifer", "Jon", "Jane")
```

# Covariance and Contravariance

# Generic hierarchy

Consider some generic class C<T>.
We can create concrete classes:

- `C<X>`
- `C<Y>`

Suppose we know:

`Y <: X`

What should we conclude?

`C<Y> <: C<X>` ❓

`C<X> <: C<Y>` ❓

```
// Y <: X
open class X(val name: String)
class Y(name: String, var age: Int): X(name)
```

```
// Box
class Box<T>(content: T)
```

Either assumption is type safe without additional information.

# Generic Hierarchy

Suppose Y <: X.  For example, we can have Y=Cat, X=Animal, i.e.

Cat <: Animal

Is it safe to assume that:

C[Cat] <: C[Animal]

```
val myCatBox: Box[Cat](Cat("Meow"))

val myAnimalBox: Box[Animal] = myCatBox

val myAnimal: myAnimalBox.content ✅
```

```
// myAnimalBox.content: Animal

myAnimalBox.content = Dog("Woof") ❌
```

No, it's not safe to assume Box[Cat] <: Box[Animal]

# Generic Hierarchy

Suppose Y <: X.  For example, we can have Y=Cat, X=Animal, i.e.

```
Cat <: Animal

Dog <: Animal
```

Is it safe to assume that:

```
Box[Animal] <: Box[Cat]
```

```
val myDog: Dog = Dog("Woof") ✅

val myAnimalBox = Box[Animal](myDog) ✅

val myCatBox: Box[Cat] = myAnimalBox ✅ 😰
```

No, it's not safe to assume `Box[Animal] <: Box[Cat]`