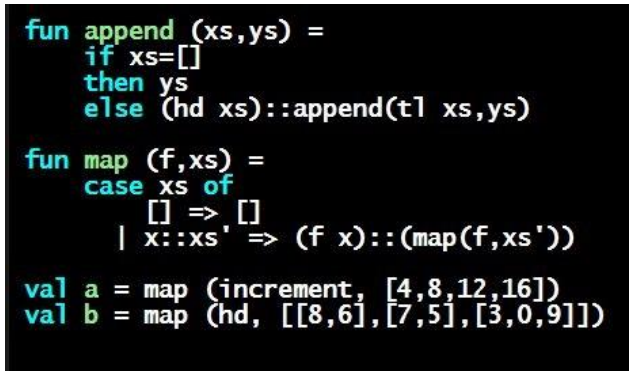

Programming Languages

Dan Grossman

Optional: Java/C# Arrays

Picking on Java (and C#)

Arrays should work just like records in terms of depth subtyping

- But in Java, if **`t1 <: t2`**, then **`t1[] <: t2[]`**
- So this code type-checks, surprisingly

```
class Point { … }
class ColorPoint extends Point { … }
…
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr); // !
  return cpt_arr[0].color; // !
}
```

Why did they do this?

- More flexible type system allows more programs but prevents fewer errors
  - Seemed especially important before Java/C# had generics
- Good news: despite this “inappropriate” depth subtyping
  - **`e.color`** will never fail due to there being no **`color`** field
  - Array *reads* **`e1[e2]`** always return a (subtype of) **`t`** if **`e1`** is a **`t[]`**
- Bad news: to get the good news
  - **`e1[e2]=e3`** can fail even if **`e1`** has type **`t[]`** and **`e3`** has type **`t`**
  - Array *stores* check the *run-time class* of **`e1`**'s elements and do not allow storing a supertype
  - No type-system help to avoid such bugs / performance cost

So what happens

```
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4); // can throw
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  …
  m1(cpt_arr); // "inappropriate" depth subtyping
  ColorPoint c = cpt_arr[0]; // fine, cpt_arr
    // will always hold (subtypes of) ColorPoints
  return c.color; // fine, a ColorPoint has a color
}
```

- Causes code in **`m1`** to throw an **`ArrayStoreException`**
  - Even though logical error is in **`m2`**
  - At least run-time checks occur only on array stores, not on field accesses like **`c.color`**

null

- Array stores probably the most *surprising* choice for flexibility over static checking
- But **`null`** is the most *common* one in practice
  - **`null`** is not an object; it has *no* fields or methods
  - But Java and C# let it have *any* object type (backwards, huh?!)
  - So, in fact, we do *not* have the static guarantee that evaluating **`e`** in **`e.f`** or **`e.m(…)`** produces an object that has an **`f`** or **`m`**
  - The "or **`null`**" caveat leads to run-time checks and errors, as you have surely noticed
- Sometimes **`null`** is convenient (like ML's option types)
  - But also having "cannot be **`null`**" types would be nice