The Matchable Trait

A new trait `Matchable` controls the ability to pattern match.

The Problem

The Scala 3 standard library has a type `IArray` for immutable arrays that is defined like this:

```
opaque type IArray[+T] = Array[_ <: T]
```

The `IArray` type offers extension methods for `length` and `apply`, but not for `update`; hence it seems values of type `IArray` cannot be updated.

However, there is a potential hole due to pattern matching. Consider:

```
val imm: IArray[Int] = ...
imm match
  case a: Array[Int] => a(0) = 1
```

The test will succeed at runtime since `IArray`s *are* represented as `Array`s at runtime. But if we allowed it, it would break the fundamental abstraction of immutable arrays.

Aside: One could also achieve the same by casting:

```
imm.asInstanceOf[Array[Int]](0) = 1
```

But that is not as much of a problem since in Scala `asInstanceOf` is understood to be low-level and unsafe. By contrast, a pattern match that compiles without warning or error should not break abstractions.

Note also that the problem is not tied to opaque types as match selectors. The following slight variant with a value of parametric type `T` as match selector leads to the same problem:

```
def f[T](x: T) = x match
  case a: Array[Int] => a(0) = 0
f(imm)
```

Finally, note that the problem is not linked to just opaque types. No unbounded type parameter or abstract type should be decomposable with a pattern match.

The Solution

There is a new type `scala.Matchable` that controls pattern matching. When typing a pattern match of a constructor pattern `C( ... )` or a type pattern `_: C` it is required that the selector type conforms to `Matchable`. If that's not the case a warning is issued. For instance when compiling the example at the start of this section we get:

```
> sc ../new/test.scala -source future
-- Warning: ../new/test.scala:4:12 ------------------------------------------
4 |    case a: Array[Int] => a(0) = 0
  |            ^^^^^^^^^^
  |            pattern selector should be an instance of Matchable,
  |            but it has unmatchable type IArray[Int] instead
```

To allow migration from Scala 2 and cross-compiling between Scala 2 and 3 the warning is turned on only for `-source future-migration` or higher.

`Matchable` is a universal trait with `Any` as its parent class. It is extended by both `AnyVal` and `AnyRef`. Since `Matchable` is a supertype of every concrete value or reference class it means that instances of such classes can be matched as before. However, match selectors of the following types will produce a warning:

- Type `Any`: if pattern matching is required one should use `Matchable` instead.
- Unbounded type parameters and abstract types: If pattern matching is required they should have an upper bound `Matchable`.
- Type parameters and abstract types that are only bounded by some universal trait: Again, `Matchable` should be added as a bound.

Here is the hierarchy of top-level classes and traits with their defined methods:

```
abstract class Any:
  def getClass
```

```
  def isInstanceOf
  def asInstanceOf
  def ==
  def !=
  def ##
  def equals
  def hashCode
  def toString

trait Matchable extends Any

class AnyVal extends Any, Matchable
class Object extends Any, Matchable
```

`Matchable` is currently a marker trait without any methods. Over time we might migrate methods `getClass` and `isInstanceOf` to it, since these are closely related to pattern-matching.

`Matchable` and Universal Equality

Methods that pattern-match on selectors of type `Any` will need a cast once the Matchable warning is turned on. The most common such method is the universal `equals` method. It will have to be written as in the following example:

```
class C(val x: String):

  override def equals(that: Any): Boolean =
    that.asInstanceOf[Matchable] match
      case that: C => this.x == that.x
      case _ => false
```

The cast of `that` to `Matchable` serves as an indication that universal equality is unsafe in the presence of abstract types and opaque types since it cannot properly distinguish the meaning of a type from its representation. The cast is guaranteed to succeed at run-time since `Any` and `Matchable` both erase to `Object`.

For instance, consider the definitions

```
opaque type Meter = Double
def Meter(x: Double): Meter = x

opaque type Second = Double
def Second(x: Double): Second = x
```

Here, universal `equals` will return true for

```
Meter(10).equals(Second(10))
```

even though this is clearly false mathematically. With multiversal equality one can mitigate that problem somewhat by turning

```
import scala.language.strictEquality
  Meter(10) == Second(10)
```

into a type error.

‹ Kind P... The @t... ›