

What problem does using the type pattern  
when the type is parameterized cause?

What's the exception to this rule

It produces counterintuitive results, since the parameters are erased.

The exception is Arrays because of implementation with Java arrays.

```
case a:Array[String] => //will work as  
expected.
```

Give the two uses of `\_ \` syntax.

- They force y to be interpreted as an identifier instead of as a keyword.
- They force the identifier starting with a lower case letter to be interpreted as a constant instead of as a variable.

Describe wildcard patterns.

They match every value, just like variable pattern, but it does not introduce a variable to refer to that value.

Describe variable patterns.

They match every value.

The variable can then be used on the right side of the case clause.



What code does the case keyword get you?

- Factory method with the name of the class.
- All arguments of the class parameter list get `val` prefix.
- Natural implementations of `toString`, `hashCode`, and `equals`.

What's the point of case classes?

They allow pattern matching on objects without huge amounts of boilerplate.

How does Scala restrict patterns?

It restricts them to be linear, meaning a pattern variable may appear only once in a pattern.

Combine patterns.

Use |

```
city match {  
  case: "Boca" | "Boca Raton" => //...  
}
```



What will typing a wildcard match **not** do?

It will not restrict the matches. It will just invite a runtime type mismatch error.

```
case _: Int => //will still match String, etc.
```

Describe constant patterns.

They match values that are equal to the constant with respect to  $==$ .

What is match selector now typically called?

The scrutinee.

How can you bind a variable to a pattern, so that the matched value has a name?

This even works if...

*name* @ *pattern*

e.g.,

UnOp("abs", e @ UnOp("abs", \_)) =>

This even works if it already has a name, or on wildcards.

e @ a =>

e @ \_ =>



List the available patterns.

Constant

Variable

Wildcard

Constructor

Sequence

Tuple

Typed patterns

What values can be used in constant patterns?

Any literal, val, or singleton can be used.

Describe pattern guards.

They come after a pattern and start with if.

An arbitrary boolean expression typically referring to variables in the pattern:

```
case n: Int if 0 < n => ...
```

Suppress warnings on non-exhaustive matches.

**Add @unchecked to the selector expression.**

```
(e: @unchecked) match {  
  //non-exhaustive match  
}
```



Describe sequence patterns.

They match sequences like List or Array just like case classes.

They also allow `_*` as the last element of a pattern, which matches any number of elements, including 0.

Write case sequence as partial functions.

```
val second: List[Int] => Int = {  
  case x::y::_ => y  
}
```

Compiler will emit non-exhaustive (i.e. incomplete function) warning. If you want this, do:

```
val second: PartialFunction[List[Int],Int] = {  
  case x::y::_ => y  
}
```

Give two ways to force constant interpretation of an identifier in a match expression.

- Prefix with an object qualifier like `this.pi` or `obj.pi`.
- Enclose the name with back ticks `\pi\`.

What are the differences between Java switch and Scala match?

- Match is an expression (i.e. always results in a value).
- Match does not fall through.
- If no pattern matches, a `MatchError` is thrown.
- Match can be applied to **Any** scrutinee.



Describe constructor patterns.

They use what appears to be a constructor. Each argument is itself a pattern, which allows nesting, variables, etc.

Where can you use patterns outside of match expressions?

- Non-function variable definitions.
- Creating partial functions with case sequences.
- In generators of `for` expressions.
- In definitions in `for` expressions.

How can you use patterns in generators?

As bound val. If there is no match, then dropped.

```
for (Some(fruit) <- results) {...}  
where results is List[Option[String]].
```

How can you use patterns in non-function variable definitions?

Unpack tuples or case classes.

```
val (a,b) = someTuple2  
val BinOp (op,left,right) = exp
```



What do typed patterns replace?

```
asInstanceOf[T], asInstanceOf[T] : T
```

They replace type tests and type casts, and just give a type to a variable. You can ignore type parameters with an `_`.

```
case s: String => //  
case m: Map[_,_] => //
```

How does the pattern matcher differentiate between variables and constants?

A simple name starting with a lower case letter is a variable,  
and all other references are constants.