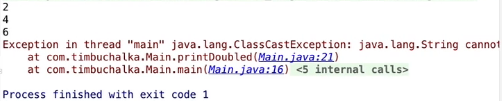
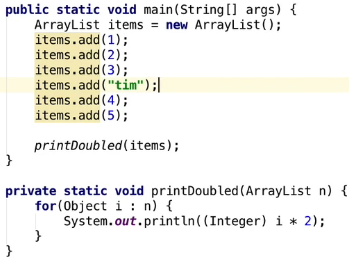
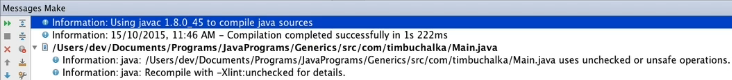
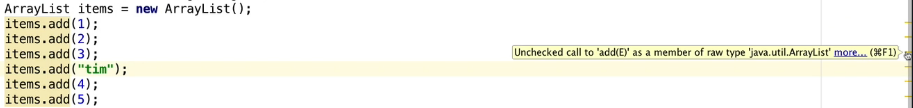
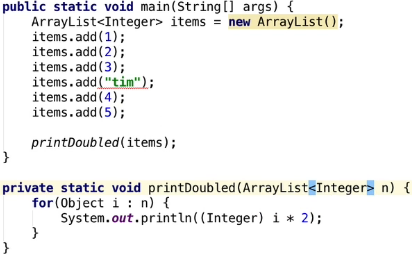
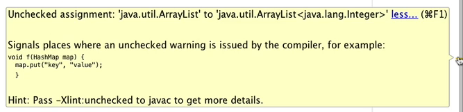
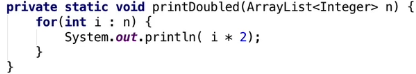
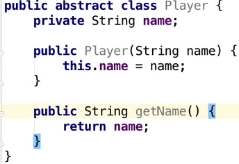
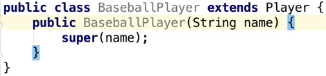
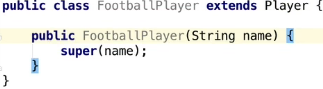
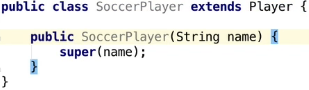
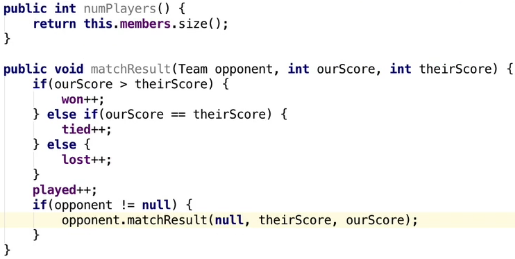
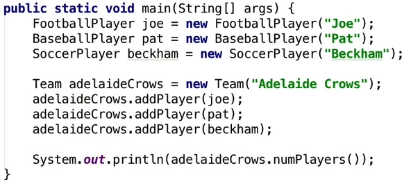
**Generics Introduction**  
\* We’ve already used generics in the course and they are incredibly useful.  
\* Just as we can create methods that take arguments in Java and we can replace the parameters that we declare for the method, generics allow us to create classes, interfaces and methods that take types as parameters called **TYPE PARAMETERS**.  
\* You’ve seen it in use in previous videos when we created ArrayLists and also LinkedLists.  
\* Let’s start with an ArrayList example that doesn’t use generics. We’re gonna create an ArrayList populated with integers and a method that prints the contents of the ArrayList doubled.  
  
\* It’s using autoboxing automatically, converting the primitive type into an Integer.  
\* We used Object in the for loop because we didn’t actually specify what the object was above and we also had to cast it as an Integer because we need to tell Java what kind of object that is.  
\* But technically, the ArrayList as we’ve created it, can actually contain anything.  
=> That’s actually more than a minor inconvenience, because it also completely breaks the compiler’s type checking.  
\* When it comes to building the project, you don’t see any errors, just warning.  
\* View => Tool Windows => Messages:  
  
\* It also tells us when we highlight this here:  
  
\* So basically the Java compiler and obviously IntelliJ by default know how to provide these warnings because they’re looking at your code and they know that ArrayList is a **GENERIC type** but **we’re using it without specifying a type parameter, in other words using it without generics**. What we haven’t done is sort of said with our ArrayList what type of objects we’re going to be storing in the ArrayList. So the ArrayList as we used it here is a **RAW type**.  
\* Prior to Java 1.5, this was the only way to code, there was no generics. And when Java 1.5 came out, they introduced the generics which we’re about to talk about. And they left this old way of doing things in purely for backwards compatibility with code that’s written from previous versions. Now you should almost never use these raw types anymore.  
\* **It’s certainly not recommended to use these raw types, and because type safety is so easy to implement these days in the current versions of Java, I highly recommend you do that**.

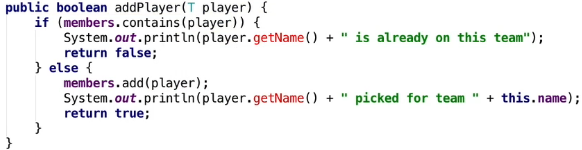
\* So when we provide a type parameter to a generic type, this is called a **PARAMETERIZED type**. So I’ve actually specified the type using angle brackets **<>**.  
  
\* Now it’s throwing an error, not allowing us to make that mistake.  
\* We still see the warning:  
  
\* If we open the detail, it gives us an example:  
  
\* We can fix the warning by specifying the type when we create a new ArrayList:  
  
\* There are called the **PARAMETERIZED types**.  
\* **In Java 1.7+ it will actually give us a warning that we should use just the <> a diamond**.  
(File -> Project Structure… -> Project Language level)  
  
=>  
  
\* In the for loop we can now remove the (Integer) cast and we can use Integer or int instead of Object:  
  
\* Now let’s look at creating our own generic classes.

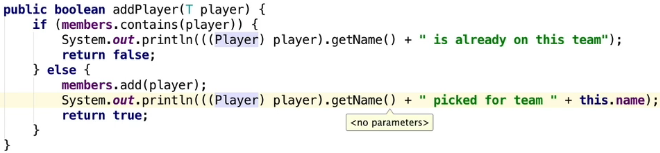
**Our Generics Class**  
\* I just want to go through a couple of brief points that are the reason we’re using generics.  
\* It’s commonly understood in programming terms that the earlier a bug is spotted in the code, the easier and ultimately cheaper it is to fix and that’s because there’s less resources and less man power invested to fix that error. So if a bug appeaers in production code, it has to be fixed. There’s no doubts about it. For most bugs, some minor bugs that perhaps you wouldn’t need to fix, but ultimately if it’s in production mode, they must be fixed and so as one is fixing it, the code itself must be then fully tested to ensure that the actual fixing didn’t in fact introduce another bug. And then of course finally the code contaning the fix has to be re-deployed and uploaded.

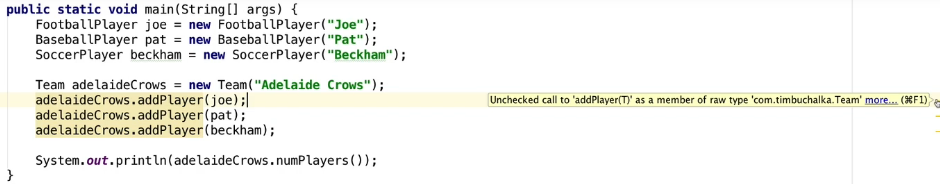
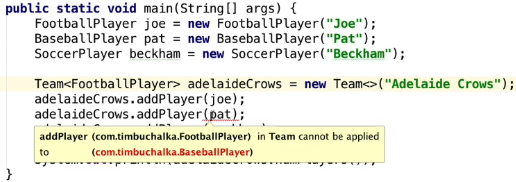
\* And so any systems, so that’s any other apps that rely on that program, they too have to be tested to ensure that the fix hasn’t broken interoperability between the two systems.  
\* If the bug itself is spotted in testing, the effort is actually reduced slightly but there’s still a requirement for any tests that previously passed to be performed again.  
\* And of course, if the bug is spotted at compile time, no one else actually notices, except you, the programmer.  
\* That’s the reasons why you want to get these errors fixed early in your code. You don’t wanna find all these bugs later in the process because particularly with larger applications and when you’re involving other resources, other people, it starts costing a lot of money and if nothing else, the programmers’ salaries and time to fix those errors.  
\* So basically what I’m getting at here is that it follows that anything we can do to pick up these bugs at compile time when we’re actually writing our code for the first time, is well worth the effort to get right.  
\* Let’s create our own generic class.  
\* A very basic abstract class for a player:  


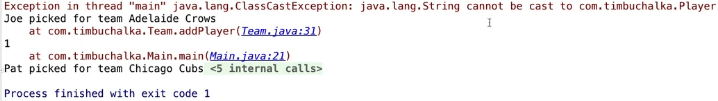
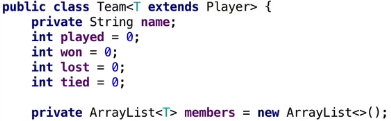
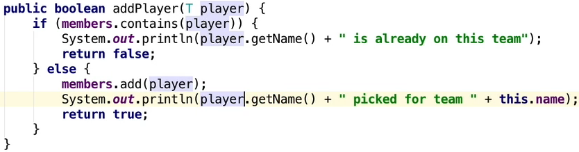
  
  
  
\* There’s nothing generic abou those at this stage.  
\* Now we’re going to create a Team class, and in that Team class, add the ability to add players.  
\* We’re gonna create an ArrayList that’s using generics for our Player objects. Player is our abstract class that all the other three player classes extend from.  
  
\* **Thanks to using an abstract class, we have 1 addPlayer() method and we can add any of the 3 types of players**.  
\* **And we’re using generics here for the ArrayList definition of type Player**.  
  
\* Instead of having to pass whether we won or lost we get the method within this Team class to actually determine that.  
\* We’re doing it this way by calling matchResult() for the opponent so that we can also update the opponent’s score and when we call this opponent matchResult(), we’re passing null this time so it doesn’t get executed again at the end. So this way we’re saving our results for our team but we’re also saving the results in the opponent’s team as well, saving the result in their separate object.   
  
\* This is one way of ranking, just keeping it simple. You could actually implement this dependent on the actual class itself but we’re just keeping it simple for now.  
\* Now we can start creating various types of teams and adding players.  


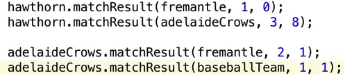
\* In the next vieo we’ll actually tackle this problem: this is the problem of actually making sure that a particular team will only accept a player of the correct type. So in this case, if we actually specified team, then we only allow the correct people who are part of that team, in other words a FootballPlayer in this scenario for the “Adelaide Crows” to be accepted as part of that team.

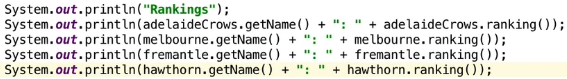
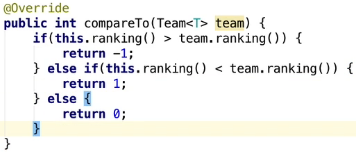
**Our Generics Class Part 2**  
\* There’s an obvious problem and that’s that the first time in this case our BaseballPlayer Pat the first time he actually runs onto the field and tries to hit the football with his bat, he’s probably gonna be red carded and sent off the field. Our Team class is actually allowing this to add any kind of Player to the team and there’s no actual check to see whether they’re the correct type of player.  
\* There’s a few solutions to this but one solution would be to create three different Team classes, so in other words a FootballTeam, a BaseballTeam and a SoccerTeam and ensure that they only accpet the correct Player type. But in that case, the code’s gonna be largely identical. So we’d be duplicating code and it’s usually never an efficient way to code to actually duplicate codes so I don’t recommend that. And of course that’s not to say we couldn’t extend the Team class to create 3 different team types within there, so sort of have private classes for each of those team types. But then if we did that, we need to implement functionality that’s unique to each team type in that code as well.  
=> So a common code to do things like add a player, play a game, record a score, etc., would be in the base Team class. But fortunately, Java’s got **Generics** and it **enables us to really specify type when we’re creating our class**. And this is exactly what we want here. So let’s modify Team to be a **generic class**.  
\* At the moment it’s obviously accepting a generic Player there in the ArrayList, also in the addPlayer() method. **First, we’re going to modify the class declaration to include the type parameter**. To do that we use the **diamond** as we did before. <T> that’s gonna indicate that there’s gonna be a type there.  
\* We’ll do a similar thing with our ArrayList so instead of <Player>, we’ll use <T> as well.  
  
\* And we’ll also do the same with our addPlayer() method.  
  
\* **So again we’re changing the type so we can make this more generic and it’s going to work for any type of player**.

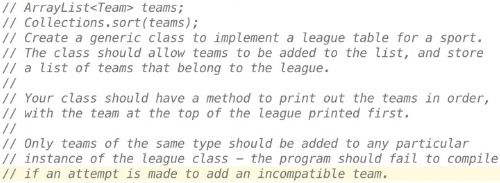
\* How this basically works is when we instantiate a class, the T will be replaced automatically by Java with the actual class we’re using, with the real type.  
\* **Notice that we’re getting an error in our addPlayer() and the reason is that we specify that its type is T, which is a parameterized type as we’ve said before, and because this type’s not known until we actually instantiate a class - in other words at run time - there’s actually no way for Java to know that an object of type T in this case really does have a .getName() method**. So the only sensible thing that a compilar can do in this situation is to flag an error.  
\* For now we’re going to cast it to Player when we’re referencing it:  
  
\* Now it’s a solution but it’s a pretty ugly cast, in the sense that we’re invoking a method on a cast object and we need parenthesis around the cast before we can use getName(). But do not fear, it’s only temporary, very soon I’ll show you how you can remove the need to do this.

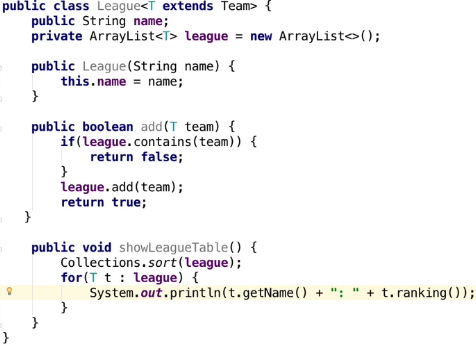
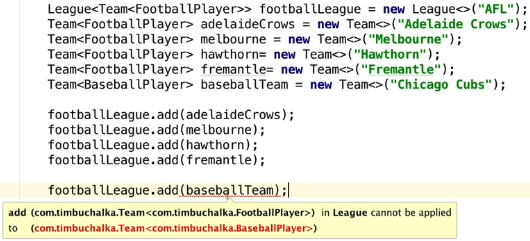
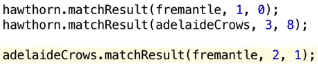
\* If we look at some of these warnings in our Main.java, we’re getting an unchecked call to addPlayer(T) as a member of raw type com.timbuchalka.team. I talked about raw types in the previous video and what they were and how to get around them and you also saw why we shouldn’t be using them.  
  
\* That said the program is still going to run, but poor old “Pat” will still be given a red card, so in other words we’re not validating the type of player against the particular type of team.  
\* But we can fix that quite easily by specifying what type of team our football team variable should be. So Team now has the ability to accept the type argument.  
  
\* But there are still some problems here and at the moment the type paramater T in our Team class can be supplied any type except primitive types such as int, so in other words any Object. So we’re not doing any validation of that.   
  
\* The problem with this is gonna be the cast when we actually run this.

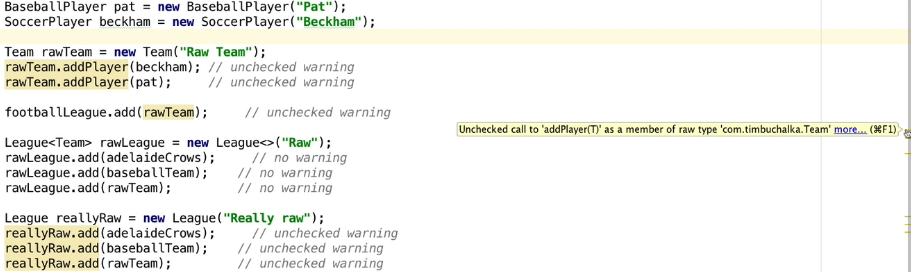
  
\* So what we really need is some sort of way to validate the type of class when we’re using it for this team. We need the ability to restrict the type that can be provided for this T argument in the Team class.  
\* Good thing about that is that Java also provides a mechanism for restricting the types that can be used as type arguments and actually call these **Bounded Type Parameters**.  
\* You can provide them with an upper bound.  
\* Let’s change the declaration of our team class first.  
\* **<T extends Player>**  
  
\* Now it gives us an error when we try to use String in there:  
  
\* **We told Java that the type parameter we’ll accept for this class when using generics, is going to be any type that extends from Player or subclass of Player**.  
\* **Player is set to be the upper bound of T**.  
\* Now we can remove the cast.  
  
\* I want to discuss a few points.  
\* 1) **An argument passed for a type parameter can either be a Class or an Interface**.  
\* 2) **Interfaces themselves can also specify type parameters**. So in other words you can create instances of a class that implements an interface and then you can ensure that the interface methods you implement act on a specific type of object or objects.  
\* And we’ve seen that in previous examples when we used the list String for example, this was an Interface that takes a generic type parameter. And the usuage is the same as it is for class.  
\* We also used a single bound when we specified the team top parameter, so restricting teams to being created from objects that will inherit from the Player class or subclass of Player only.  
\* Java does allow multiple bounds, so you can have class T extends Player and Coach and Manager for example to allow multiple instances of Player, Coach or Manager to be added to the team. But with that in mind, the normal inheritance implements rules still apply, so in other words you can only extend from the single class but multiple interefaces. So if you want to do this with more than one class, you’ll need to use interfaces for those other types as well.  
\* Another thing to keep in mind is **if you’re specifying multiple bounds, then the class must be listed first because otherwise Java’s gonna raise an error when you try to compile it. In other words class first and then interfaces**.

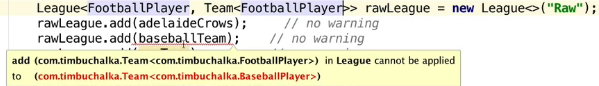
\* This would be the syntax:  
  
\* Player would the the class and Coach and Manager would have to be interfaces.  
\* And obviously in the case of moddable types, you have to then include code to check what type of objects you’re dealing with before invoking its methods.  
\* The important thing here to remember is that there’s no relationship between List<FootballPlayer> and List<SoccerPlayer> if we created Lists for these 2, because even though they are a subtype of this Player so they do have a common parent, but the common parent in this case is List:  
  
\* The bottom line is that probably a lot of the time with generics you’ll probably be doing something like this where you’ll only be extending or using one class and maybe an interface. But if you do need to use multiple bounds, you’ll need to specify the & and then obviously the interfaces as well.  
\* One more thing we need to look at => so the Team class has got fields, check how many games were won, tied or lost, but we haven’t actually made use of them yet. We do include a matchResult() method so let’s make some calls on it now in Main.java just to record the game results before we discuss the method more.  
  
\* Let’s have the method give us more feedback:  
  
\* Notice that we’re printing this message only for the initial team. This makes sense but because we’re printing the name of the opponent, we get an error if we take this part of the message outside of the check for opponnet being null.  
\* There is a problem - the Adelaide Crows drew against the Chicago Cubs but the Adelaide Crows played football and of course Cubs played baseball so that’s rather an odd match result.  
=> So what we really wanna do here is make sure that only the correct team type can be passed to the matchResult() method as well instead of any object of type Team. So we need to use a generic type.  
  


**Our Generics Class Part 3**  
\* The Team class includes a very simple ranking.  
  
\* Let’s print it out.  
  
\* One thing we can’t do is compare teams to see which would be the highest in the league table.  
\* Let’s implement something that’s very useful, an interface called **Comparable**.  
\* Now we need to implement the **compareTo()** function.  
\* We’re adding diamonds again because we don’t want to compare football to baseball teams.  
\* The compareTo() method takes an object and returns a negative number if this is less than the object, 0 if they’re equal, and a positive number if this is greater than the object.  
  
\* We’ve actually used the compareTo() method when we implemented our own LinkedList. Many of the Java classes implement this interface Comparable, including the String and Integer classes. That’s how these objects can be sorted as a result of doing that.  
  
  
  
\* The reason we’ve implemented the Comparable interface is that it’s going to help you with the challenge to come in the next video.   
=> **If you’ve got a list of objects that implement Comparable, what you can do is you can quickly sort the list by using the static** **Collections.sort()** **method of the** **Collections class**.  
\* It sorts it using the compareTo() function.

**Generics Challenge**  


  
\* **The reason that Collections.sort() works is because within Team we’ve written the compareTo() method, it implements Comparable. This will be used automatically by the Collections framework when we’re sorting**.  
\* We defined League to be a parameterized type that’s bound by the Team class. So in other words a Team or any subclasses would be allowed as the type argument when we create an instance of this League. And the parameterized type is used in the declaration of the ArrayList up here as well.  
  
\* Now let’s print the results.  
  


  
\* Before I finish I wanna just go through the warnings about using **RAW TYPES** because it’s really important to understand why we actually don’t wanna do that.  
\* RAW types are allow in order to allow the legacy code, that’s code prior to Java 1.5, the code written before the GENERICS was introduced to Java to allow that to work.  
\* But you can’t rely on the compiler to give you an error, you actually get a warning, IntelliJ will also warn you but the code will compile.  
  
\* So we’re not actually defining the types as we did before.  
\* So the code will still run but we’ll get those warnings.  
\* It’s allowing us to add any kind of team to the league.  
\* That’s all because we’re using the RAW types, not the generics.  
\* You wanna be using GENERICS whenever you can, because it makes your code cleaner, less likely to have crashed and easier to debug essentially because the errors that you’re getting are gonna be there at compile time and not obviously when the code’s in production when you’re trying to fix these weird bugs.  
\* Without going into it any further, we could actually mitigate some of these problems, to at least prevent the rawLeague declaration from compiling.  
=> We could go back to our League class.  


  
\* But it’s not an ideal solution.