Generics 11

Chapter Topics

- Understanding the relationship between generic types, parameterized types, and raw types
- Declaring generic types (normal classes, record classes, enum types and interfaces) and parameterized types
- Use of the diamond operator (`<>`) when creating objects of generic classes
- Using the restricted keyword `var` to declare local generic type references
- Extending generic types
- Understanding the folly of mixing generic code and legacy code
- Understanding the significance of unchecked warnings on type-safety
- Understanding subtype relationships for wildcards
- Understanding type hierarchy for wildcard parameterized types
- Understanding widening and narrowing reference conversions in type hierarchy of wildcard parameterized types
- Understanding restrictions on set and get operations when using references of wildcard parameterized types
- Using bounded type parameters
- Understanding how to implement a generic class that is also `Iterable`
- Understanding wildcard capture
- Programming with wildcard parameterized types
- Writing generic methods and constructors
- Understanding the implications of type erasure
- Understanding how overloading and overriding work with generics
- Understanding reifiable and non-reifiable types, and their role in generics
- Understanding the limitations and restrictions that generics place on instance tests, casting, arrays, varargs, and exception handling

Java SE 17 Developer Supplementary Topics	
[12.3] Use generics, including wildcards. ❍ *Note that this is a supplementary objective in the description of the Java SE 17 Developer exam.*	§11.1, p. 565 *to* §11.4, p. 579
Java SE 11 Developer Exam Objectives	

[5.1] Use generics, including wildcards	§11.1, p. 565 *to* §11.13, p. 623

Introduction of generics in Java was a major step in the evolution of the language. Extending the language with generics provides even stronger type checking at compile time and allows implementing generic algorithms, especially for managing *containers* of data, like the collections discussed in **Chapter 15**.

Generics in Java is a large topic with some subtle details. Extending the language with generics has consequences for many aspects of the language. Familiarity with topics covered in this chapter will be beneficial in both using and implementing generic types.

The basics of generics are covered in §11.1, §11.2, and §11.3, which introduce the terminology, the underlying concepts, and the motivation for using generics for implementing collections.

On a first reading, §11.4 to §11.8 should be read to become familiar with wildcards and the flexibility they provide in defining parameterized types, and with generic methods and constructors. The reader should come back to these topics for an in-depth study when the need arises.

For an even deeper dive into generics, we suggest studying the topics from §11.9 onward: wildcard capture and wildcard parameterized types, implications of type erasure, and restrictions on generic types.

11.1 Introducing Generics

Generics allow classes and interfaces, as well as methods and constructors, to be *parameterized* with *type information*. An *abstract data type* (ADT) defines both the *types* of objects and the *operations* that can be performed on these objects. Generics allow us to specify the types used by the ADT so that the *same* definition of an ADT can be used on *different* types of objects.

Generics in Java are a way of providing type information in ADTs so that the compiler can guarantee type-safety of operations at runtime. Generics are implemented as compile-time transformations, with negligible impact on the JVM. The generic type declaration is compiled once into a single Java class file, and the use of the generic type is checked against this file. Also, no extraneous Java class files are generated for each use of the generic type.

The primary benefits of generics are increased language expressiveness with improved type-safety, resulting in improved robustness and reliability of code. Generics avoid verbosity of using casts in many contexts, thus improving code clarity. Since the compiler

guarantees type-safety, this eliminates the necessity of explicit type checking and casting at runtime.

One major goal when introducing generics in Java has been backward compatibility with legacy code (i.e., non-generic code). Interoperability with legacy code and the lack of generic type information at runtime largely determine how generics work in Java. Many of the restrictions on generics in Java can be attributed to these two factors.

Generics are used extensively in implementing the Java Collections Framework. An overview of **Chapter 15** on collections and maps is therefore recommended as many of the examples in this chapter make use of generic types from this framework.

Before the introduction of generics in Java, a general implementation of a collection maintained its objects by using references of the type `Object`. The bookkeeping of the actual type of the objects fell on the client code. **Example 11.1** illustrates this approach. It implements a *self-referential data structure* called a *node*. Each node holds a data value and a reference to another node. Such data structures form the basis for building *linked data structures*.

Example 11.1 *A Legacy Class*

Click here to view code image

```
class LegacyNode {
  private Object    data;    // The value in the node
  private LegacyNode next;   // The reference to the next node.
  LegacyNode(Object data, LegacyNode next) {
    this.data = data;
    this.next = next;
  }
  public void       setData(Object obj)      { this.data = obj; }
  public Object     getData()                { return this.data; }
  public void       setNext(LegacyNode next) { this.next = next; }
  public LegacyNode getNext()                { return this.next; }
  @Override public String toString() {
    return this.data + (this.next == null? "" : ", " + this.next);
  }
}
```

The class `LegacyNode` can be used to create a linked list with arbitrary objects:

Click here to view code image

```
LegacyNode node1 = new LegacyNode(4, null);       // 4 --> null
LegacyNode node2 = new LegacyNode("July", node1); // "July" --> 4 --> null
```

Primitive values are encapsulated in corresponding wrapper objects. If we want to retrieve the data from a node, the data is returned via an `Object` reference:

Click here to view code image

```
Object obj = node2.getData();
```

In order to access type-specific properties or behavior of the fetched object, the reference value in the `Object` reference must be converted to the right type. To avoid a `ClassCastException` at runtime when applying the cast, we must make sure that the object referred to by the `Object` reference is of the right type. All that can be accomplished by using the `instanceof` pattern match operator:

Click here to view code image

```
if (obj instanceof String str) {
  System.out.println(str.toUpperCase()); // Method specified in the String class.
}
```

The approach outlined above places certain demands on how to use the class `LegacyNode` to create and maintain linked structures. For example, it is the responsibility of the client code to ensure that the objects being put in nodes are of the same type. Implementing classes for specific types of objects is not a good solution. First, it can result in code duplication, and second, it is not always known in advance what types of objects will be put in the nodes. Generic types offer a better solution, where one generic class is defined and specific reference types are supplied each time we want to instantiate the class.

11.2 Generic Types and Parameterized Types

We first introduce the basic terminology and concepts relating to generics in Java. Note that the discussion here on generic and parameterized types also applies to enum types (§5.13, p. 287) and record classes (§5.14, p. 299).

Generic Types

A *generic type* is a reference type that defines a *list of formal type parameters* or *type variables* that must be provided before it can be used as a type. **Example 11.2** declares a generic type which, in this case, is a *generic class* called `Node<E>` that allows nodes of specific types to be maintained. It has only one formal type parameter, `E`, that represents the type of the data in a node.

```
class Node<E> {
...
}
```

The formal type parameter `E` does not explicitly specify a type, but serves as a placeholder for a type to be defined in an invocation of the generic type. The formal type parameters of a generic type are specified within angle brackets, <>, immediately after the class name. A type parameter is an unqualified identifier. If a generic class has several formal type parameters, these are specified as a comma-separated list, `<T1, T2, ... , Tn>`. It is quite common to use one-letter names for formal type parameters; a convention that we will follow in this book. For example, `E` is used for the type of elements in a collection, `K` and `V` are used for the type of the keys and the type of the values in a map, and `T` is used to represent an arbitrary type.

As a starting point for declaring a generic class, we can begin with a class where the `Object` type is utilized to generalize the use of the class. In **Example 11.2**, the declaration of the generic class `Node<E>` uses `E` in all the places where the type `Object` was used in the declaration of the class `LegacyNode` in **Example 11.1**. From the declaration of the class `Node<E>`, we can see that the formal type `E` is used like a reference type in the class body: as a field type at (1), as a return type at (5), and as a parameter type in the methods at (4) to (8). Use of the class name in the generic class declaration is parameterized by the type parameter ((2), (6), (7)), with one notable exception: The formal type parameter is *not* specified after the class name in the constructor declaration at (3). Which actual reference type the formal type parameter `E` represents is not known in the generic class `Node<E>`. Therefore, we can only call methods that are inherited from the `Object` class on the field `data`, as these methods are inherited by all objects, regardless of their object type. One such example is the call to the `toString()` method in the method declaration at (8).

The scope of the type parameter `E` of the generic type includes any non-static inner classes, but excludes any static member types—the parameter `E` cannot be accessed in static context. It also excludes any nested generic declarations where the same name is redeclared as a formal type parameter. Shadowing of type parameter names should be avoided.

Example 11.2 *A Generic Class for Nodes*

Click here to view code image

```
class Node<E> {
  private E            data;    // Data                              (1)
  private Node<E>      next;    // Reference to next node            (2)
  Node(E data, Node<E> next) {                                    // (3)
    this.data = data;
    this.next = next;
  }
  public void    setData(E data)       { this.data = data; }   // (4)
  public E       getData()             { return this.data; }   // (5)
  public void    setNext(Node<E> next) { this.next = next; }   // (6)
  public Node<E> getNext()             { return this.next; }   // (7)
```

```
  @Override public String toString() {                          // (8)
    return this.data.toString() +
           (this.next == null ? "" : ", " + this.next.toString());
  }
}
```

Some Restrictions on the Use of Type Parameters in a Generic Type

A constructor declaration in a generic class cannot specify the formal type parameters of the generic class in its constructor header after the class name:

Click here to view code image

```
class Node<E> {
  ...
  Node<E>() { ... }                         // Compile-time error!
  ...
}
```

A formal type parameter cannot be used to create a new instance, as it is not known which concrete type it represents. The following code in the declaration of the `Node<E>` class would be illegal:

Click here to view code image

```
E ref = new E();                            // Compile-time error!
```

A formal type parameter is a *non-static type*. It cannot be used in a static context, for much the same reason as an instance variable cannot be used in a static context: It is associated with objects. The compiler will report errors at (1), (2), and (3) in the code below:

Click here to view code image

```
class Node<E> {
  private static E e1;                          // (1) Compile-time error!
  public  static E oneStaticMethod(E e2) {      // (2) Compile-time error!
    E e3;                                       // (3) Compile-time error!
    System.out.println(e3);
  }
  // ...
}
```

Parameterized Types

A *parameterized type* (also called a *type instance*) is an *invocation* or *instantiation* of a generic type that is a specific usage of the generic type where the formal type parameters are replaced by *actual type parameters*. Analogy with method declarations and method invocations can be helpful in understanding the relationship between generic types and parameterized types. We pass actual parameters in a method invocation to execute a method. In the case of a generic type invocation, we pass actual *type* parameters in order to instantiate a generic type.

We can declare references and create objects of parameterized types, and call methods on these objects, in much the same way as we use non-generic classes.

Click here to view code image

```
Node<Integer> intNode = new Node<Integer>(2020, null);
```

The actual type parameter `Integer`, explicitly specified in the declaration statement above, binds to the formal type parameter `E` in **Example 11.2**. The compiler treats the parameterized type `Node<Integer>` as a *new type*. The parameterized type `Node<Integer>` constrains the generic type `Node<E>` to `Integer` objects, thus implementing homogenous nodes with `Integer`s. The reference `intNode` can only refer to a `Node` of `Integer`. The node created can only be used to store an object of this concrete type.

Methods can be called on objects of parameterized types:

Click here to view code image

```
Integer iRef = intNode.getData();            // Integer object with int value 2020
```

In the method call above, the actual type parameter is determined from the type of the reference used to make the call. The type of the `intNode` reference is `Node<Integer>`; therefore, the actual type parameter is `Integer`. The method header is `Integer getData()`, meaning that the method will return a value of type `Integer`. The compiler checks that the return value can be assigned. As the compiler guarantees that the return value will be an `Integer` and can be assigned, no explicit cast or runtime check is necessary. The compiler actually inserts the necessary cast. Here are some more examples of calling methods of parameterized types:

Click here to view code image

```
intNode.setData(2020);                                  // Ok.
intNode.setData("TwentyTwenty");                        // (1) Compile-time error!
```

```
intNode.setNext(new Node<Integer>(2019, null));    // (2020, (2019, null))
intNode.setNext(new Node<String>("Hi", null));     // (2) Compile-time error!
```

In the method calls shown above, the compiler determines that the actual type parameter is `Integer`. The method signatures are `setData(Integer)` and `setNext(Node<Integer>)`. As expected, we get a compile-time error when we try to pass an argument that is not compatible with the parameter type in the method declarations; for example, at (1) and (2). The parameterized types `Node<Integer>` and `Node<String>` are two *unrelated* types. The compiler reports any inconsistent use of a parameterized type so that errors can be caught earlier at compile time and the use of explicit casts in the source code is minimized, as evident from (3) and (4), respectively.

Click here to view code image

```
Node<String> strNode = new Node<String>("Hi", null);
intNode = strNode;                          // (3) Compile-time error!
String str = strNode.getData();             // (4) No explicit cast necessary.
```

The Diamond Operator (<>)

In the object creation expression of the `new` operator, the actual type parameter was explicitly specified after the class name—in contrast to the constructor declaration.

Click here to view code image

```
Node<String> lst = new Node<String>("Hi", null); // Explicit actual type parameter
```

The actual type parameters can be omitted, but not the angle brackets (`<>`), if the compiler can infer the actual type parameters from the context of the object creation expression. The angle brackets with no actual parameters (`<>`) are commonly referred to as the *diamond operator*.

Click here to view code image

```
Node<String> lst = new Node<>("Hi", null); // Actual type parameter inferred.
```

In the object creation expression above, the compiler performs automatic type inference to infer that the actual type parameter in the expression must be `String`. The compiler is able to infer the actual type parameter from the type information of the constructor call arguments.

Click here to view code image

```
new Node<>(null, null);  // Actual type parameter: Object.
```

In the code below, the compiler uses the type information of the variable on the left-hand side to infer the actual type parameter of the object creation expression, thereby ensuring compatibility with the *target type* on the left-hand side of the assignment.

Click here to view code image

```
Node<String>  strNode = new Node<>(null, null); // Actual type parameter: String.
Node<Integer> intNode = new Node<>(null, null); // Actual type parameter: Integer.
Node<Number>  numNode = new Node<>(null, null); // Actual type parameter: Number.
Node<Number>  lstNode = new Node<>(2021, null); // Actual type parameter: Number.
```

In the last declaration, the `int` value `2021` is boxed into an `Integer` object that can be assigned to a reference of its superclass `Number`. In other words, the signature of the constructor call is `Node<Number>(Number, Node<Number>)`.

Given the following scenario:

Click here to view code image

```
// (1) Method declaration with parameterized type as formal parameter.
void find(Node<String> node) { /* ... */ }
...
// (2) Method call where actual argument uses diamond operator.
find(new Node<>(null,null));  // Actual type parameter: Object or String?
```

The compiler takes the target type (`Node<String>`) in the method declaration into consideration, correctly inferring the actual type parameter to be `String`, in order for the actual and the formal parameters in the call to be assignment compatible.

A single diamond operator must replace the entire actual type parameter list in the object creation expression. In the first declaration below, the compiler infers that the actual type parameter list is `<String, List<Integer>>`.

Click here to view code image

```
HashMap<String, List<Integer>> map = new HashMap<>();
HashMap<String, List<Integer>> map = new HashMap<String,<>>(); // Error!
```

If the actual type parameters are not specified and the diamond operator is omitted, the compiler issues an *unchecked conversion warning*—that is, the code will compile, but all bets are off at runtime. Below, the reference type `Node` in the object creation expression is

interpreted as a *raw type*. Implications of interoperability between generic types and raw types are discussed on p. 575.

Click here to view code image

```
Node<String> rawNode = new Node("Hi", null); // Unchecked conversion warning!
```

The diamond operator can be used to instantiate an anonymous class (see p. 633).

Click here to view code image

```
new Node<>("Hi", null) {/* ... */}; // Parameterized type: Node<String>
new Node("Hi", null)   {/* ... */}; // Raw type: Node
```

Parameterized Local Variable Type Inference

Consider the four local variable declarations shown below. The first three declarations are equivalent, as they create a `Node<String>` object whose `data` and `next` fields are `null`. In declaration (1), since the type `String` is explicitly specified in the object creation expression, the actual type parameter is deduced to be `String`. In declaration (2), the diamond operator is specified in the object creation expression, but the compiler is able to infer that the actual type parameter is `String` from the left-hand side of the declaration—that is, the compiler considers the context of the whole declaration. In the case of the parameterized local variable declarations with `var`, both the actual type parameters and the parameterized type denoted by `var` are inferred from the object creation expression.

Click here to view code image

```
// Parameterized local variable declarations:
Node<String> node1 = new Node<String>(null, null); // (1) Node of String
Node<String> node2 = new Node<>(null, null);       // (2) Node of String
var node3 = new Node<String>(null, null);          // (3) Node of String
var node4 = new Node<>(null, null);                // (4) Node of Object
```

In declaration (3), as `String` is explicitly specified in the object creation expression, the actual type parameter is inferred to be `String` and the parameterized type of `node3` denoted by `var` is inferred to be `Node<String>`.

However, in declaration (4), the diamond operator is used in the object creation expression. In this case, the actual type parameter is inferred to be `Object` and the parameterized type of `node4` denoted by `var` is inferred to be `Node<Object>`. Adequate type information should be provided in the object creation expression when declaring parameterized local variables with `var` in order to avoid unexpected types being inferred for the actual parameter types and the parameterized type denoted by `var`.

Generic Interfaces

Generic types also include generic interfaces, which are declared analogous to generic classes. The specification of formal type parameters in a generic interface is the same as in a generic class. **Example 11.3** declares a generic interface that defines the reference type `IMonoLink<E>` for objects that store a data value of type `E`.

Example 11.3 *A Generic Interface and Its Implementation*

Click here to view code image

```
interface IMonoLink<E> {
  void         setData(E data);
  E            getData();
  void         setNext(IMonoLink<E> next);
  IMonoLink<E> getNext();
}

class MonoNode<E> implements IMonoLink<E> {
  private E            data;    // Data
  private IMonoLink<E> next;    // Reference to next node                  (1)

  MonoNode(E data, IMonoLink<E> next) {                                  // (2)
    this.data = data;
    this.next = next;
  }

  @Override public void setData(E data) { this.data = data; }
  @Override public E    getData()       { return this.data; }
  @Override public void setNext(IMonoLink<E> next) { this.next = next; } // (3)
  @Override public IMonoLink<E> getNext()          { return this.next; } // (4)
  @Override public String toString() {
    return this.data.toString() + (this.next == null? "" : ", " + this.next);
  }
}
```

A generic interface can be implemented by a generic (or a non-generic) class:

Click here to view code image

```
class MonoNode<E> implements IMonoLink<E> {
  // ...
}
```

Note that the construct `<E>` is used in two different ways in the class header. The first occurrence of `<E>` *declares* `E` to be a type parameter, and the second occurrence of `<E>` *parameterizes* the generic interface with this type parameter. The declare-before-use rule

also applies to type parameters. The version of the `MonoNode` class in **Example 11.3** differs from the `Node` class in **Example 11.2** at (1), (2), (3), and (4). These changes were necessary to make the `MonoNode<E>` class compliant with the `IMonoLink<E>` interface.

A generic interface can be parameterized in the same way as a generic class. In the code below, the reference `strNode` has the parameterized type `IMonoLink<String>`. It is assigned the reference value of a node of inferred type `MonoNode<String>`. The assignment is legal, since the parameterized type `MonoNode<String>` is a subtype of the parameterized type `IMonoLink<String>`:

Click here to view code image

```
IMonoLink<String> strNode2 = new MonoNode<>("Bye", null);
System.out.println(strNode2.getData());                    // Prints: Bye
```

As with non-generic interfaces, generic interfaces cannot be instantiated either:

Click here to view code image

```
IMonoLink<String> strNode3 = new IMonoLink<>("Bye", null); // Compile-time error!
```

Example 11.4 shows a non-generic class implementing a generic interface. The generic interface `IMonoLink<E>` is parameterized by a *concrete type*, namely, `Lymph`. The type `LymphNode` is a subtype of the parameterized type `IMonoLink<Lymph>`, as it implements the methods of the generic interface `IMonoLink<E>` in accordance with the concrete type parameter `Lymph`.

The Java standard library contains many examples of generic interfaces. The two interfaces `java.lang.Comparable<E>` and `java.util.Comparator<E>` are discussed in detail in §14.4, p. 761, and §14.5, p. 769, respectively. The Java Collections Framework also includes many examples of generic interfaces, such as `Collection<E>`, `List<E>`, `Set<E>`, and `Map<K,V>` (Chapter 15, p. 781).

Example 11.4 *A Non-Generic Class Implementing a Generic Interface*

Click here to view code image

```
// File: LymphNode.java
class Lymph { /*... */ }

public class LymphNode implements IMonoLink<Lymph> {
  private Lymph            body;
  private IMonoLink<Lymph> location;
  @Override public void  setData(Lymph obj) { body = obj; }
  @Override public Lymph getData()          { return body; }
```

```
    @Override public void  setNext(IMonoLink<Lymph> loc) { this.location = loc; }
    @Override public IMonoLink<Lymph> getNext()          { return this.location; }
  }
```

Extending Generic Types

A non-final generic type can be extended. **Example 11.5** shows that the generic interface `IBiLink<E>` extends the generic interface `IMonoLink<E>`, and that the generic class `BiNode<E>` extends the generic class `MonoNode<E>` and implements the generic interface `IBiLink<E>` (see **Figure 11.1**).

Click here to view code image

```
interface IBiLink<E> extends IMonoLink<E> {
  // ...
}
class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  // ...
}
```

The compiler checks that the formal type parameters of the superclass in the `extends` clause can be resolved. In the case above, the formal type parameter `E`, which is specified for the subclass, is also used as the type parameter for the superclass and is used to constrain the interface to the same type parameter. This dependency ensures that an invocation of the subclass will result in the same actual type parameter being used by the superclass and for the interface.

Click here to view code image

```
BiNode<Integer> intBiNode = new BiNode<>(2020, null, null);
MonoNode<Integer> intMonoNode = intBiNode;        // (1)
Integer iRef = intMonoNode.getData();             // Integer with int value 2020
MonoNode<Number> numMonoNode = intBiNode;         // (2) Compile-time error!
```

The assignment at (1) is type-safe, as the parameterized class `BiNode<Integer>` is a subtype of the parameterized class `MonoNode<Integer>`. It is important to note that the superclass and the subclass are parameterized with the *same* type parameter; otherwise, the subtype relationship between the superclass and the subclass does not hold. We get a compile-time error at (2) because the parameterized class `BiNode<Integer>` is *not* a subtype of the parameterized class `MonoNode<Number>`. Subtype relationships for generic types are discussed in a later section (**p. 579**).

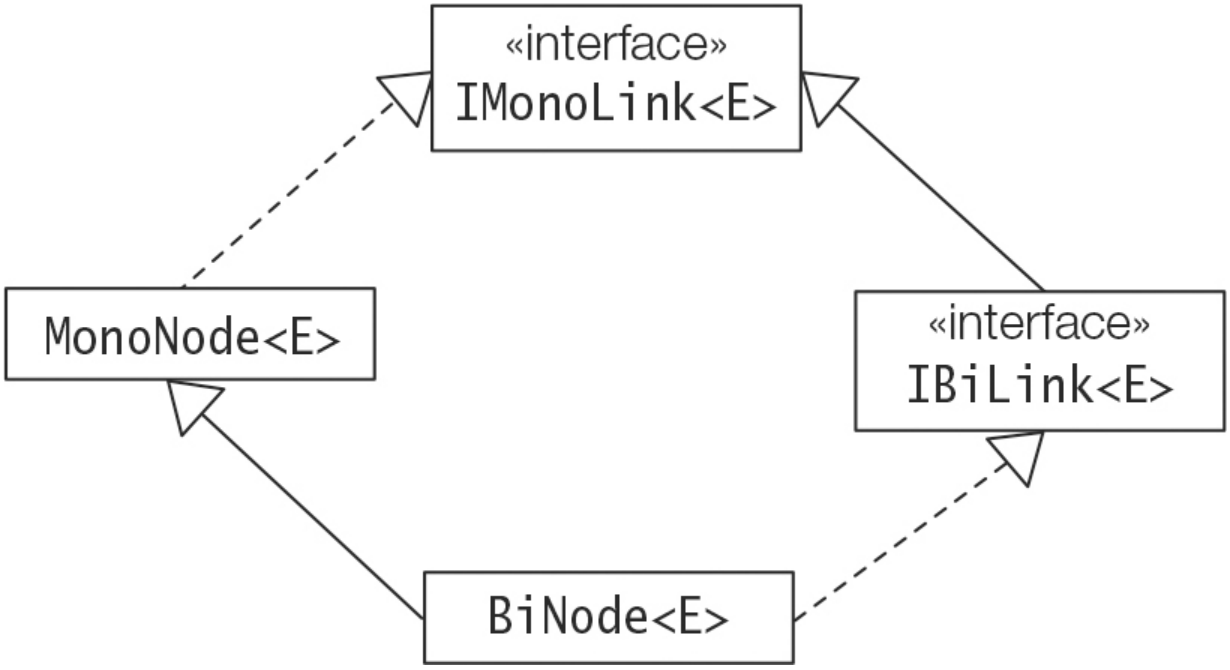

Figure 11.1 *Extending Generic Types*

Example 11.5 *Extending Generic Types*

Click here to view code image

```
interface IBiLink<T> extends IMonoLink<T> {
  void       setPrevious(IBiLink<T> previous);
  IBiLink<T> getPrevious();
}

class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  private IBiLink<E>  previous;    // Reference to previous node

  BiNode(E data, IBiLink<E> next, IBiLink<E> previous) {
    super(data, next);
    this.previous = previous;
  }
  @Override public void setPrevious(IBiLink<E> previous) {
    this.previous = previous;
  }
  @Override public IBiLink<E> getPrevious() { return this.previous; }
  @Override public String toString() {
    return (this.previous == null? "" : this.previous + ", ") +
            this.getData() +
           (this.getNext() == null? "" : ", " + this.getNext());
  }
}
```

Example 11.5 showed examples of generic types being extended to create new generic subtypes. We can extend a non-generic type to a generic subtype as well:

Click here to view code image

```
class AbstractNode { /* ... */ }                    // A non-generic supertype
class SimpleNode<E> extends AbstractNode { /* ... */ }// A generic subtype
```

We can also extend concrete parameterized types to specialized non-generic subtypes:

Click here to view code image

```
class IntegerBiNode extends BiNode<Integer> {         // A non-generic subtype
  IntegerBiNode(Integer data, IntegerBiNode next, IntegerBiNode previous) {
    super(data, next, previous);
  }
  //...
}
```

Note that a subtype can inherit only one parameterization of the same generic interface supertype. Implementing or extending a parameterized type fixes the parameterization for the subtype and its supertypes. In the declaration below, the subtype `WeirdNode<E>` tries to implement the interface `IMonoLink<Integer>`, but at the same time, it is a subtype of the interface `IMonoLink<E>` which the superclass `Mono-Node<E>` implements:

Click here to view code image

```
class WeirdNode<E> extends MonoNode<E> implements IMonoLink<Integer> { // Error!
  //...
}
```

There is great flexibility in extending reference types, but care must be exercised to achieve the desired result.

Raw Types and Unchecked Warnings

A generic type without its formal type parameters is called a *raw type*. The raw type is the supertype of all parameterized types of the generic type. For example, the raw type `Node` is the supertype of the parameterized types `Node<String>`, `Node<Integer>`, and `Node<Node<String>>`. The last parameterized type is an example of a *nested parameterization*. It means that a node of this type has a node of type `Node<String>` as data.

A parameterized type (e.g., `Node<String>`) is *not* a class. Parameterized types are used by the compiler to check that objects created are used correctly in the program. The parameterized types `Node<String>`, `Node<Integer>`, and `Node<Node<String>>` are all represented at runtime by their raw type `Node`. In other words, the compiler does *not* create a new class for each parameterized type. Only one class (`Node`) exists that has the name of the generic class (`Node<E>`), and the compiler generates only one class file (`Node.class`) with the Java bytecode for the generic class.

Only reference types (excluding array creation and enumerations) can be used in invocations of generic types. A primitive type is not permitted as an actual type parameter, the reason being that values of primitive types have different sizes. This would require different code being generated for each primitive type used as an actual type parameter, but there is only one implementation of a generic class in Java.

Generics are implemented in the compiler only. The JVM is oblivious about the use of generic types. It does not differentiate between `Node<String>` and `Node<Integer>`, and just knows about the class `Node`. The compiler translates the generic class by a process known as *type erasure*; meaning that information about type parameters is erased and casts are inserted to make the program type-safe at runtime. The compiler guarantees that casts added at compile time never fail at runtime, when the program compiles without any *unchecked warnings*.

It is possible to use a generic class by its raw type only, like a non-generic class, without specifying actual type parameters for its usage. **Example 11.6** illustrates mixing generic and non-generic code. The compiler will issue an unchecked warning if such a use can be a potential problem at runtime. Such usage is permitted for backward compatibility with legacy code, but is strongly advised against when writing new code.

The assignment at (5) in **Example 11.6** shows that it is always possible to assign the reference value of a parameterized type to a reference of the raw type, as the latter is the supertype of the former. However, the raw type reference can be used to violate the type-safety of the node at runtime, as shown at (6). Calling a method on a node using the raw type reference results in an *unchecked call warning* by the compiler. In this particular case, a `String` is set as the data of an `Integer` node.

Click here to view code image

```
...
Node<Integer> intNode = new Node<>(2020, null);
Integer iRef = intNode.getData();       // Integer object with int value 2020
...
Node rawNode = intNode;          // (5) Assigning to raw type always possible.
rawNode.setData("BOOM");         // (6) Unchecked call warning!
intNode = rawNode;               // (7) Unchecked conversion warning!
iRef = intNode.getData();        // (8) ClassCastException!
iRef = rawNode.getData();        // (9) Compile-time error!
```

Assigning the reference value of a raw type to a reference of the parameterized type results in an *unchecked conversion warning* from the compiler, as shown at (7). If the node referred to by the raw type reference is not of type `Integer`, using it as a node of type `Integer` can lead to problems at runtime, as shown at (8). The assignment at (8) is only type compatible, not type-safe, as its type-safety is compromised at (6) as explained above.

A `ClassCastException` is thrown at runtime, since an `Integer` was expected, but a `String` was returned by the `getData()` method.

The assignment at (9) does not compile because of type mismatch: Without the generic type information, the compiler infers that the call on the `getData()` method using the raw type reference `rawNode` can only return an `Object`, whereas the type of the variable on the left-hand side is `Integer`.

The class `Preliminaries` in **Example 11.6** is shown compiled with the non-standard option `-Xlint:unchecked`. The compiler recommends using this option when non-generic and generic code are mixed in this way. The program compiles in spite of the unchecked warnings, and can be executed. But all guarantees of type-safety are off in the face of unchecked warnings. See also §11.11, p. 613, which provides details on translation of generic code by type erasure.

Example 11.6 *Unchecked Warnings*

Click here to view code image

```
// A client for the generic class Node<E> in Example 11.2, p. 568.
public class Preliminaries {
  public static void main(String[] args) {

    Node<Integer> intNode = new Node<>(2018, null);
    Integer iRef = intNode.getData();        // Integer object with int value 2018
    intNode.setData(2020);                   // Ok.
//  intNode.setData("TwentyTwenty");         // (1) Compile-time error!
    intNode.setNext(new Node<>(2019, null)); // (2020, (2019, null))
//  intNode.setNext(new Node<>("Hi", null)); // (2) Compile-time error!

    Node<String> strNode = new Node<>("Hi", null);
//  intNode = strNode;              // (3) Compile-time error!
    String str = strNode.getData(); // (4) No explicit cast necessary.

    Node rawNode = intNode;         // (5) Assigning to raw type always possible.
    rawNode.setData("BOOM");        // (6) Unchecked call warning!
    intNode = rawNode;              // (7) Unchecked conversion warning!
    iRef = intNode.getData();       // (8) ClassCastException!
//  iRef = rawNode.getData();       // (9) Compile-time error!
  }
}
```

Compiling the program:

Click here to view code image

```
>javac -Xlint:unchecked Preliminaries.java
Preliminaries.java:16: warning: [unchecked] unchecked call to setData(E) as a
member of the raw type Node
    rawNode.setData("BOOM");        // (6) Unchecked call warning!

                    ^
  where E is a type-variable:
    E extends Object declared in class Node
Preliminaries.java:17: warning: [unchecked] unchecked conversion
    intNode = rawNode;              // (7) Unchecked conversion warning!

              ^
  required: Node<Integer>
  found:    Node
2 warnings
```

Running the program:

Click here to view code image

```
>java Preliminaries
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot
 be cast to java.lang.Integer
        at Preliminaries.main(Preliminaries.java:18)
```

11.3 Collections and Generics

Before the introduction of generics in Java 1.5, a collection in the Java Collections Framework could hold references to objects of any type. For example, any object which is an instance of the `java.lang.Object` class or its subclasses could be maintained in an instance of the `java.util.ArrayList` class, which implements the `java.util.List` interface.

Click here to view code image

```
List wordList = new ArrayList();     // Using non-generic types.
wordList.add("two zero two zero");   // Can add any object.
wordList.add(2020);
//...
Object element = wordList.get(0);    // Always returns an Object.
//...
if (element instanceof String str) { // Runtime check to avoid ClassCastException.
  // Use reference str.
}
```

The client of a collection has to do most of the bookkeeping with regard to using the collection in a type-safe manner: which objects are put in the collection and how objects retrieved are used. Using the `Object` class as the element type allows the implementation of the collection class to be specific, but its use to be generic. An `ArrayList` is a specific implementation of the `List` interface, but usage of the class `ArrayList` is generic with regard to any object.

Using a generic collection, the compiler provides the type-safety, and the resulting code is less verbose.

Click here to view code image

```
List<String> wordList = new ArrayList<>();          // Using a specific type.
wordList.add("two zero two zero");                  // Can add strings only.
wordList.add(2020);                                 // Compile-time error!
//...
String element = wordList.get(0);                   // Always returns a String.
//...
```

Runtime checks or explicit casts are not necessary now. Generic types allow the implementation of the collection class to be generic, but its use to be specific. The generic type `ArrayList<E>` is a generic implementation of the `List<E>` interface, but now the usage of the parameterized type `ArrayList<String>` is specific, as it constrains the generic type `ArrayList<E>` to strings.

11.4 Wildcards

In this section, we discuss how using wildcards can increase the expressive power of generic types. But first we examine one major difference between array types and parameterized types. The generic class `Node<E>` used in this subsection is defined in **Example 11.2, p. 568**.

The Subtype Covariance Problem with Parameterized Types

The following three declarations create three nodes of `Integer`, `Double`, and `Number` type, respectively.

Click here to view code image

```
Node<Integer> intNode    = new Node<>(2020,null);          // (1)
Node<Double>  doubleNode = new Node<>(3.14,null);          // (2)
Node<Number>  numNode    = new Node<>(2021, null);         // (3)
```

In the declaration at (3), the signature of the constructor call is `Node(Integer, null)`. The formal type parameter `E` of the generic class `Node<E>` is bound to the actual type pa-

rameter `Number`—that is, the signature of the constructor is `Node(Number, Node<Number>)`. Since the type `Integer` is a subtype of the type `Number`, and `null` can be assigned to any reference, the constructor call succeeds.

In the method calls at (4) and (5) below, the method signature in both cases is `setData(Number)`. The method calls again succeed, since the actual parameters are of types `Double` and `Integer`, which are subtypes of `Number`:

Click here to view code image

```
numNode.setData(10.5);                                  // (4)
numNode.setData(2022);                                  // (5)
```

However, the following calls do *not* succeed:

Click here to view code image

```
numNode.setNext(intNode);                               // (6) Compile-time error!
numNode = new Node<Number>(2030, doubleNode);           // (7) Compile-time error!
```

The actual type parameter at (6) is determined to be `Number`. The generic class `Node<E>` is thus parameterized with the class `Number`. The compiler complains that the method `setNext(Node<Number>)` in the parameterized class `Node<Number>` is *not* applicable for the actual argument (`Node<Integer>`) at (6)—that is, the method signature `setNext(Node<Number>)` is *not* compatible with the method call signature `setNext(Node<Integer>)`. The compiler also complains at (7): The constructor signature `Node(Number, Node<Number>)` is *not* applicable for the arguments `(int, Node<Double>)`. The problem is with the second argument at (7). We cannot pass an argument of type `Node<Integer>` or `Node<Double>` where a parameter of type `Node<Number>` is expected. The following assignments will also not compile:

Click here to view code image

```
numNode = intNode;                                      // (8) Compile-time error!
numNode = doubleNode;                                   // (9) Compile-time error!
```

The reason for the compile-time errors is that `Node<Integer>` and `Node<Double>` are *not* subtypes of `Node<Number>`, although `Integer` and `Double` are subtypes of `Number`. In the case of arrays, the array types `Integer[]` and `Double[]` *are* subtypes of the array type `Number[]`. The subtyping relationship between the individual types carries over to corresponding array types. This type relationship is called *subtype covariance* (see **Figure 11.2**). This relationship holds for arrays because the element type is available at runtime, and can be checked. If the subtype covariance were allowed for parameterized type, it

could lead to problems at runtime, as the element type would not be known and cannot be checked, since it has been erased by the compiler.

Click here to view code image

```
numNode = intNode;                  // If this assignment was allowed,
numNode.setData(25.5);              // the data could be corrupted,
Integer iRef = intNode.getData();   // resulting in a ClassCastException!
```

Therefore, the *subtype covariance* relationship does *not* hold for parameterized types that are instantiations of the same generic type with different actual type parameters, regardless of any subtyping relationship between the actual type parameters. The actual type parameters are *concrete* types (e.g., `Integer`, `Number`), and therefore, the parameterized types are called *concrete parameterized types*. Such parameterized types are totally unrelated. As an example from the Java Collections Framework, the parameterized type `Map<Integer, String>` is not a subtype of the parameterized type `Map<Number, String>`.

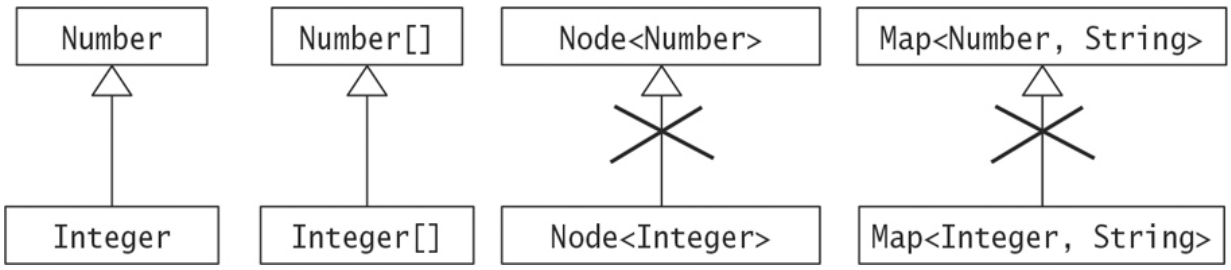

Figure 11.2 *No Subtype Covariance for Parameterized Types*

Wildcard Types

Wildcard types are type parameters defined using the *wildcard* symbol `?`. The wildcard `?` by itself represents *all* types. The parameterized type `List<?>` represents a list of all types, whereas the concrete parameterized type `List<Integer>` only represents a list of `Integer`. In other words, a *wildcard type* can represent *many types*. Therefore, a parameterized type that has wildcard types as actual type parameters can represent a family of types, in contrast to a concrete parameterized type that only represents itself. The wildcard types provided in Java represent four subtype relationships that are summarized in **Table 11.1**.

Wildcard types provide the solution for increased expressive power to overcome the limitations discussed earlier when using generics in Java, but introduce limitations of their own as to what operations can be carried out on an object using references of wildcard types. We will use the class `Node<E>` in **Example 11.2**, **p. 568**, as a running example to discuss the use of wildcard types.

Table 11.1 *Summary of Subtyping Relationships for Generic Types*

Name	Syntax	Semantics	Description
Subtype covariance	`? extends Type`	Any subtype of `Type` (including `Type`)	*Bounded wildcard with upper bound*
Subtype contravariance	`? super Type`	Any supertype of `Type` (including `Type`)	*Bounded wildcard with lower bound*
Subtype bivariance	`?`	All types	*Unbounded wildcard*
Subtype invariance	`Type`	Only type `Type`	*Type parameter/argument*

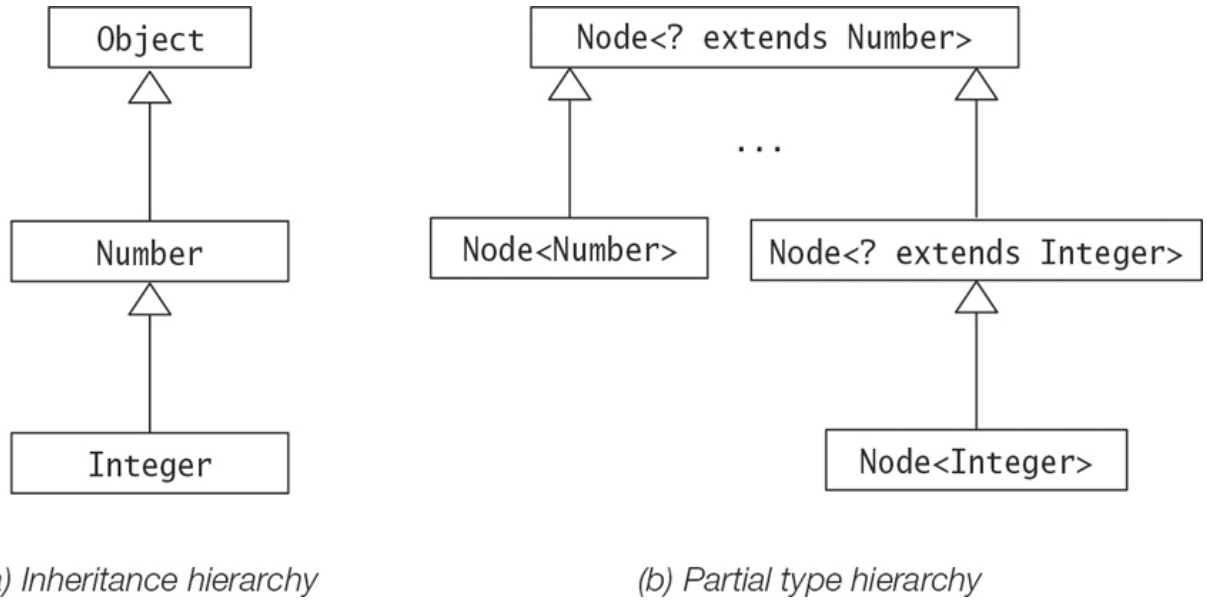

Figure 11.3 *Partial Type Hierarchy for* `Node<? extends Number>`

The wildcard type `? extends Number` denotes all subtypes of `Number`, and the parameterized type `Node<? extends Number>` denotes the family of invocations of `Node<E>` for types that are subtypes of `Number`. **Figure 11.3** shows a partial type hierarchy for the parameterized type `Node<? extends Number>`. Note that the parameterized type `Node<? extends Integer>` is a subtype of the parameterized type `Node<? extends Number>`, since the wildcard type `? extends Integer` represents all subtypes of `Integer`, and these are also subtypes of `Number`.

Click here to view code image

```
Node<? extends Integer> intSubNode = new Node<Integer>(100, null);
```

```
Node<? extends Number> numSupNode = intSubNode;
```

Subtype Contravariance: `? super Type`

The wildcard type `? super Type` represents all *supertypes* of `Type` (including `Type` itself). The wildcard type `? super Type` is called a *lower bounded wildcard* with `Type` representing its *lower bound*.

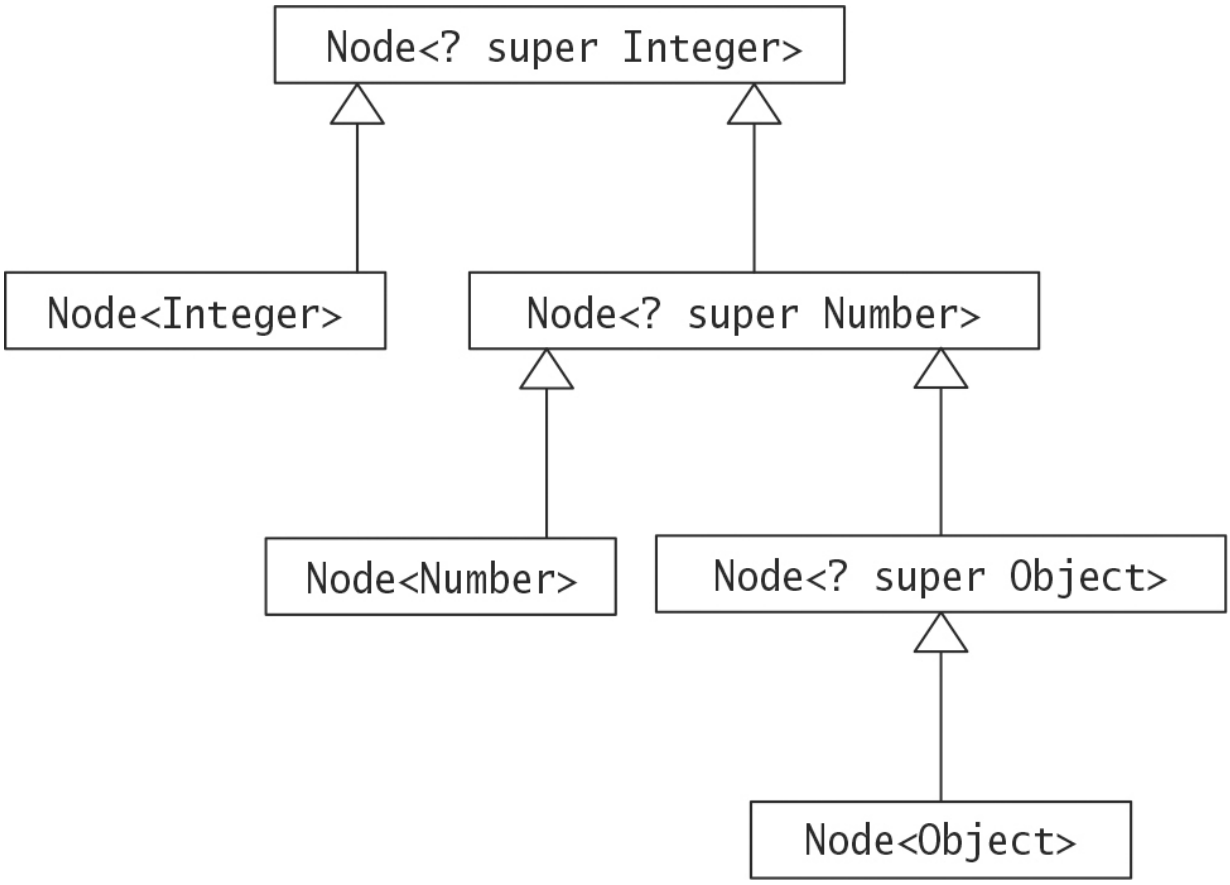

Figure 11.4 *Partial Type Hierarchy for* `Node<? super Integer>`

The wildcard type `? super Integer` denotes all supertypes of `Integer`, and the parameterized type `Node<? super Integer>` denotes a family of invocations of `Node<E>` for types that are supertypes of `Integer`. **Figure 11.4** shows a partial type hierarchy for the parameterized type `Node<? super Integer>`. Note that the parameterized type `Node<? super Number>` is a *subtype* of the parameterized type `Node<? super Integer>`, since the wildcard type `? super Number` represents all supertypes of `Number`, and these are also supertypes of `Integer`.

Click here to view code image

```
Node<? super Number> numSupNode = new Node<Number>(100, null);
Node<? super Integer> numIntSupNode = numSupNode;
```

Subtype Bivariance: `?`

As mentioned earlier, the wildcard type `?` represents *all* types. The wildcard type `?` is called the *unbounded wildcard*, since it has no bounds as do the other two wildcard types. By definition, it represents both the upper and the lower bounded wildcards for any bound.

The parameterized type `Node<?>` denotes the family of invocations of `Node<E>` for any type—that is, denotes a `Node` of any kind, and is therefore the supertype of all invocations of `Node<E>` (see also **Figure 11.5**, **p. 583**, and **§11.5**, **p. 584**).

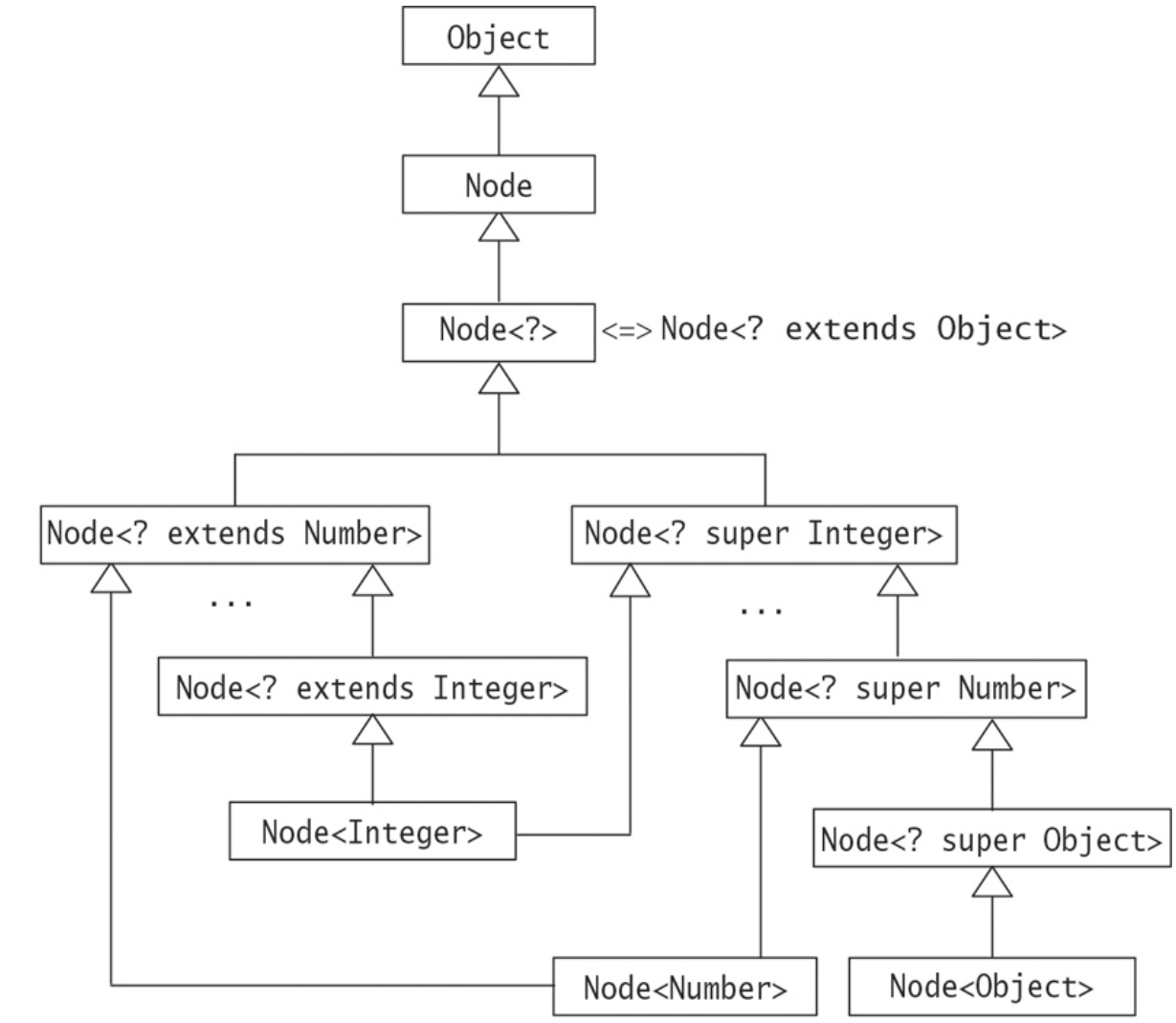

Figure 11.5 *Partial Type Hierarchy for Selected Parameterized Types of* `Node<E>`

Subtype Invariance: `Type`

When a concrete type `Type` is used as an actual type parameter in a parameterized type, it represents `Type` itself. Since `Type` can be *any* concrete type, it is called an *unbounded*

type parameter. The concrete parameterized type `Node<Integer>` represents the invocation of `Node<E>` for the concrete actual type parameter `Integer`. As we have seen earlier, there is no subtype covariance relationship between concrete parameterized types, but there is such a relationship between bounded parameterized types and concrete parameterized types (see also **Figure 11.3** and **Figure 11.4**).

Let us recapitulate the basic terminology before proceeding further. A generic type can specify one or more formal type parameters. A parameterized type is an invocation of a generic type, supplying the required actual type parameters. An actual type parameter can be a wildcard type (possibly bounded) or a concrete type. A concrete type is either a non-generic type or a parameterized type that has concrete types as parameters.

Some Restrictions on Wildcard Types

Wildcards *cannot* be used in instance creation expressions:

Click here to view code image

```
Node<?> anyNode = new Node<?>(2020, null);                 // Compile-time error!
Node<? extends Integer> extIntNodeA
                = new Node<? extends Integer>(0, null);    // Compile-time error!
Node<? extends Integer> extIntNodeB = new Node<Integer>(0, null);  // OK
```

The actual type parameter in the constructor call must be a concrete type. Creating instances of wildcard parameterized types is analogous to instantiating interface types; neither can be used in object creation expressions.

Wildcards *cannot* be used in the *header* of reference type declarations. Supertypes in the `extends` and `implements` clauses cannot have wildcards.

Click here to view code image

```
class QuestionableNode<?>  { /* ... */ }                          // Not OK.
class SubNode extends Node<?> { /* ... */ }                       // Not OK.
interface INode extends Comparable<? extends Node<?>> { /* ... */ }  // Not OK.
class XNode implements Comparable<?>  { /* ... */ }               // Not OK.
```

However, *nested wildcards* are not a problem in a reference type declaration header or in an object creation expression:

Click here to view code image

```
class OddNode extends Node<Node<?>> implements Comparable<Node<?>> { /* ... */ }
...
Node<?> nodeOfAnyNode = new Node<Node<?>>(new Node<Integer>(2020, null), null);
```

11.5 Using References of Wildcard Parameterized Types

A wildcard type can be used to declare parameterized references—that is, references whose type is a wildcard parameterized type. In this section, we look at how such references are used in the following contexts:

- Assignment between such references
- Calling methods of generic types using such references

The generic class `Node<E>` used in this subsection is defined in **Example 11.2, p. 568**.

Generic Reference Assignment

A reference of a supertype can refer to an object of a subtype, and this substitution principle applies to parameterized types as well. Assignment compatibility is according to the type hierarchy of the parameterized types. **Figure 11.5** shows partial type hierarchy for selected parameterized types of the generic class `Node<E>`. It combines the type hierarchies from **Figure 11.3** and **Figure 11.4**. As we would expect, *widening reference conversions* according to the type hierarchy are always type-safe. All but the last assignment statement in the code below are legal. The types `Node<Number>` and `Node<Integer>` are unrelated. (The notation `B <: A` means `B` is a subtype of `A`.)

Click here to view code image

```
Node<Object>  objNode = new Node<Object>(100, null);
Node<Number>  numNode = new Node<Number>(200, null);
Node<Integer> intNode = new Node<Integer>(300, null);
Node<? extends Number> extNumNode
                       = intNode; // Node<Integer> <: Node<? extends Number>
Node<? super Integer>  supIntNode
                       = numNode; // Node<Number> <: Node<? super Integer>
supIntNode = objNode;             // Node<Object> <: Node<? super Integer>
numNode    = intNode;             // Compile-time error! Types unrelated.
```

In the code below, we get an error at (1) because the types `Node<? extends Number>` and `Node<? super Number>` are unrelated, but that is not the case for the types `Node<? extends Object>` and `Node<? super Object>` at (2). The family of types denoted by the type `Node<? super Object>` has the subtype `Node<Object>` only, which is also a subtype of the type `Node<? extends Object>`. In the assignment at (3), the type `Node<? extends Object>` is not a subtype of the type `Node<? super Object>`, but the converse is true as established at (2).

Click here to view code image

```
Node<? super Number>   supNumNode;
Node<? extends Object> extObjNode;
```

```
Node<? super Object>   supObjNode;
extNumNode = supNumNode;  // (1) Compile-time error! Types unrelated.
extObjNode = supObjNode;  // (2) Node<? super Object> <: Node<? extends Object>
supObjNode = extObjNode;  // (3) Compile-time error!
```

Narrowing reference conversion requires an explicit cast, except for the cases noted below (see also **Figure 11.5**). The raw type `Node` and the unbounded wildcard parameterized type `Node<?>` are essentially equivalent in this regard. Conversion between the two is type-safe:

Click here to view code image

```
Node    rawNode;
Node<?> anyNode;
rawNode = anyNode;  // Node <-- Node<?> is type-safe.
anyNode = rawNode;  // Node<?> <-- Node is type-safe.
```

The unbounded wildcard parameterized type `Node<?>` and the upper bounded wildcard parameterized type `Node<? extends Object>` are also essentially equivalent (see (4)), except when assigned a value of the raw type `Node` (see (5)).

Click here to view code image

```
// (4):
anyNode = extObjNode;  // Node<?> <-- Node<? extends Object> is type-safe.
extObjNode = anyNode;  // Node<? extends Object> <-- Node<?> is type-safe.

// (5):
anyNode    = rawNode;  // Node<?> <-- Node is type-safe.
extObjNode = rawNode;  // Node<? extends Object> <-- Node: Unchecked Conversion
```

Assigning a value of the raw type `Node` to a reference of the type `Node<? extends Object>` results in an *unchecked conversion warning*—which conforms to the general rule when mixing legacy and generic code: Assigning the value of a raw type to a reference of a bounded wildcard parameterized type or a concrete parameterized type results in an unchecked conversion warning, as illustrated by the examples below.

Click here to view code image

```
extNumNode = rawNode; // Node<? extends Number> <-- Node: Unchecked Conversion
intNode    = rawNode; // Node<Integer> <-- Node: Unchecked Conversion
```

For a discussion of explicit casting of parameterized references, see §11.13, p. 625. Suppressing different kinds of unchecked warnings with the

`@SuppressWarnings("unchecked")` annotation is discussed in §11.13, p. 623, and §25.5, p. 1582.

Using Parameterized References to Call Set and Get Methods

Generic classes are suitable for implementing ADTs called *collections* (also called *containers*) where the element type is usually specified by a type parameter. The Java Collections Framework is a prime example of such collections. A collection usually provides two basic operations: a *set* operation (also called a *write* or *put* operation) to add an element to the collection, and a *get* operation (also called a *read* operation) to retrieve an element from the collection. The *set* operation takes a parameter of the type `T`, where `T` is a type parameter of the generic class. The *get* operation returns a value of the type parameter `T`. The class `Node<E>` provides these two basic operations to manipulate the data in a node:

Click here to view code image

```
class Node<E> {
  private E data;
  // ...
  public void setData(E obj) { data = obj; }        // (1) Set operation.
  public E    getData()      { return data; }       // (2) Get operation.
  // ...
}
```

So far we have called these two methods using references of concrete parameterized types:

Click here to view code image

```
Node<Number> numNode = new Node<>(2020, null);
numNode.setData(2021);                      // (3) Can only set a Number.
Number data = numNode.getData();            // (4) Can only get a Number.
```

The actual type parameter in the above method calls is a *concrete type*, but what happens when we use a reference of a wildcard parameterized type that represents a family of types? For example, what if the type of the reference `numNode` is `Node<? extends Number>`? Is the method call at (3) type-safe? Is the assignment at (4) type-safe? Operations that can potentially break the type-safety are flagged as either compile-time errors or warnings. If there are warnings but no errors, the program still compiles. However, type-safety at runtime is not guaranteed.

The key to using generics in Java is understanding the implications of wildcard parameterized types in the language—and why the compiler will not permit certain operations involving wildcards, since these might break the type-safety of the program. To illustrate some of the subtleties, we compile the class in Example 11.7 successively with different

headers for the method `checkIt()`. The parameter type is different in each method header, from (h1) to (h5). The method uses three local variables `object`, `number`, and `integer` of type `Object`, `Number`, and `Integer`, respectively ((v1) to (v3)). There are three calls to the `setData()` method of the generic class `Node<E>` to set an `Object`, a `Number`, and an `Integer` as the data in the node referenced by the reference `s0` ((s1) to (s3)). There are also three calls to the `getData()` method of the generic class `Node<E>`, assigning the return value to each of the local variables ((s4) to (s6)). And finally, the last statement, (s7), tests whether the data retrieved can be put back in again.

Example 11.7 *Illustrating Get and Set Operations Using Parameterized References*

Click here to view code image

```
class WildcardAccessRestrictions {

  static void checkIt(Node s0) {                    // (h1)
//static void checkIt(Node<?> s0) {                 // (h2)
//static void checkIt(Node<? extends Number> s0) {  // (h3)
//static void checkIt(Node<? super Number> s0) {    // (h4)
//static void checkIt(Node<Number> s0) {            // (h5)
    // Local variables
    Object  object  = new Object();                 // (v1)
    Number  number  = 1.5;                          // (v2)
    Integer integer = 10;                           // (v3)
    // Method calls
    s0.setData(object);                             // (s1)
    s0.setData(number);                             // (s2)
    s0.setData(integer);                            // (s3)
    object  = s0.getData();                         // (s4)
    number  = s0.getData();                         // (s5)
    integer = s0.getData();                         // (s6)
    s0.setData(s0.getData());                       // (s7)
  }
}
```

Attempts to compile the method in **Example 11.7** with different headers are shown in **Table 11.2**. The rows are statements from (s1) to (s7) from **Example 11.7**. The columns indicate the type of the parameter `s0` in the method headers (h1) to (h5). The reference `s0` is used to call the methods. The entry **ok** means the compiler did not report any errors or any unchecked warnings. The entry **!** means the compiler did not report any errors but issued an unchecked call warning. The entry × means the compiler reported an error. In other words, we cannot carry out the operations that are marked with the entry ×.

Table 11.2 *Get and Set Operations Using Parameterized References*

Type of s0 Operation	`Node`	`Node<?>`	`Node<? extends Number>`	`Node<? super Number>`	`Node<Number>`
`s0.setData(object);`	!	×	×	×	×
`s0.setData(number);`	!	×	×	**ok**	**ok**
`s0.setData(integer);`	!	×	×	**ok**	**ok**
`object = s0.getData();`	**ok**	**ok**	**ok**	**ok**	**ok**
`number = s0.getData();`	×	×	**ok**	×	**ok**
`integer = s0.getData();`	×	×	×	×	×
`s0.setData(s0.get-Data());`	!	×	×	×	**ok**

Raw Type References

The type of the reference `s0` is the raw type `Node`. This case illustrates the non-generic paradigm of using a collection: We can put any object, but we can only get an `Object`. From **Table 11.2**, we see that we can put any object as data in a node of the raw type `Node`, but the compiler issues *unchecked call warnings*, as we are putting an object into a raw type node whose element type is not known. We can only get an `Object`, as we cannot be more specific about the data type.

Unbounded Wildcard References: `<?>`

The type of the reference `s0` is `Node<?>`. The compiler determines that the actual type parameter for each method call is the wildcard `?`—that is, any type. Obviously, we cannot set any data in a node whose element type cannot be determined. It might not be type-safe. And we cannot guarantee the type of its data either because the data type can be of any type, but we can safely read it as an `Object`. Note that we can always write a `null`, as the `null` value can be assigned to any reference.

Typical use of unbounded wildcard reference is in writing methods that treat the objects in a container as of type `Object` and make use of `Object` methods for their operation. The method below can print the data in *any* sequence of nodes, given the start node. The specific type of the data in the node is immaterial. Note also that the method is not dependent on any type parameter.

Click here to view code image

```
void printNodeDataSequence(Node<?> s0) {       // Unbounded parameterized type
  Node<?> next = s0.getNext();                 // Returns node as Node<Object>
  while (next != null) {
    Object obj = next.getData();               // Object reference.
    System.out.println(obj.toString());        // Call Object method
    next = next.getNext();
  }
}
```

Upper Bounded Wildcard References: `<? extends Type>`

The type of the reference `s0` is `Node<? extends Number>`, where `Type` is `Number`. This means that the reference `s0` refers to a node containing an object whose type is either `Number` or a subtype of `Number`, but the specific (sub)type of the object cannot always be determined at compile time. Putting *any* object, except a `null`, into such a node might not be type-safe.

The code below shows what would happen if any object was allowed to be set as data in a `Long` node via its alias `s0`. If (1), (2), or (3) were allowed, we would get a `ClassCastException` at (4) because the data could not be assigned to a `Long` reference, as the type-safety of the node `longNode` will have been compromised, either with a supertype object or an object of an unrelated type.

Click here to view code image

```
Long longInt = 20L;
Node<Long> longNode = new Node<>(longInt, null);  // Node of Long, that is
Node<? extends Number> s0 = longNode;// referenced by a Node<? extends Number> ref.
s0.setData(object);                  // If this was allowed, or             (1)
s0.setData(number);                  // if this was allowed, or             (2)
s0.setData(integer);                 // if this was allowed,                (3)
longInt = longNode.getData();        // we would get an exception here.     (4)
```

The following method call will also not compile, as the compiler cannot give any guarantees at compile time that the reference `s0` will refer to a node of `Long` at runtime:

Click here to view code image

```
s0.setData(longInt);                // Compile-time error!
```

The upper bound in the wildcard type `? extends Number` is `Number`. Therefore, the data of the node with the wildcard type `? extends Number` must be a `Number` (i.e., either an object of type `Number` or an object of a subtype of `Number`). Thus we can only safely assign the reference value returned by the *get* operation to a reference of type `Number` or a supertype of `Number`.

Lower Bounded Wildcard References: `<? super Type>`

Using a reference of type `Node<? super Number>`, where `Type` is `Number`, we can only put a `Number` or a subtype object of `Number` into the node, as such a number would also be a subtype object of any supertype of `Number`. Since we cannot guarantee which specific supertype of `Number` the node actually has, we cannot put any supertype object of `Number` in the node. The code below shows what would happen if an unrelated supertype object was put in as data in a `Node<? super Number>`. If (1) were allowed, we would get a `ClassCastException` at (2) because the data value (of a supertype) cannot be assigned to a `Number` reference (which is a subtype).

Click here to view code image

```
Node<Number> numNode = new Node<>(2020, null);
Node<? super Number> s0 = numNode;
s0.setData(object);             // (1) If this set operation was allowed,
number = numNode.getData();     // (2) we would get an exception here.
```

Since the type of the reference `s0` is `Node<? super Number>`, the reference `s0` can refer to a node containing an object whose type is either `Number` or some supertype of `Number`. When we *get* the data from such a node, we can only safely assume that it is an `Object`. Keeping in mind that a reference of a supertype of `Number` can refer to objects that are unrelated to `Number` (e.g., an `Object` reference that can refer to a `String`), if (3) were allowed in the code below, we would get a `ClassCastException` at (3):

Click here to view code image

```
Node<Object> objNode = new Node<>("Hi", null);  // String as data.
Node<? super Number> s0 = objNode;
number = s0.getData();        // (3) If allowed, we would get an exception here.
object = s0.getData();        // This is always ok.
```

Unbounded Type References: `<Type>`

The type of the reference `s0` is `Node<Number>`, where `Type` is `Number`. The actual type parameter for each method call is determined to be `Number`. Thus the type of the parame-

ter in the `setData()` method and the return value of the `getData()` method is `Number`. Therefore, we can pass the reference value of a `Number` or a subclass object of `Number` to the `setData()` method, and can assign the reference value returned by the `getData()` method to a reference of the type `Number` or a supertype of `Number`. In this case, we can put a `Number`, and get a `Number`.

Table 11.3 gives a summary of using parameterized references for *set* and *get* operations on a container. Here are some general guidelines for choosing a wildcard parameterized type:

- If we only want to *get* an element of type `E` from a container, we can use the upper bounded wildcard `? extends E` for the reference.
- If we only want to *put* an element of type `E` into a container, we can use the lower bounded wildcard `? super E` for the reference.
- If we want to both *get* and *set* elements of type `E` in a container, we can use the unbounded type `E` for the reference.

The following acronym might help to remember which parameterized references should be used to invoke *get* and *set* methods of a container: *GESS* (*Get-* `extends` *-Set-* `super`), meaning for a *get* method use reference of type `<? extends T>` and for a *set* method use reference of type `<? super T>`.

Table 11.3 *Summary of Get and Set Operations Using Parameterized References*

Type of s0 **Operation**	`Node`	`Node<?>`	`Node<? extends Number>`	`Node<? super Number>`	`Node<Number>`
`set/put/write`	*Any object*	*Cannot put anythig except* `null` *s.*	*Cannot put anything except* `null` *s.*	`Number` *or subtype*	`Number` *or subtype*
`get/read`	`Object` *only*	`Object` *only*	`Number`	`Object` *only*	`Number`

11.6 Bounded Type Parameters

In the declaration of the generic class `Node<E>`, the type parameter `E` is *unbounded*— that is, it can be any reference type. However, sometimes it may be necessary to restrict what type the type parameter `E` can represent. The canonical example is restricting that the type parameter `E` is `Comparable<E>` so that objects can be compared.

Wildcard types *cannot* be used in the header of a generic class to restrict the type parameter:

Click here to view code image

```
class CmpNode<? extends Comparable> { ... }          // Compile-time error!
```

However, the type parameter can be bounded by a *constraint* as follows:

Click here to view code image

```
class CmpNode<E extends Comparable<E>> {             // E is bounded.
  // ...
}
```

In the constraint `<E extends Comparable<E>>`, `E` is *bounded* and `Comparable<E>` is the *upper bound*. This is an example of a *recursive type bound*. The declaration above states that the actual type parameter when we parameterize the generic class `CmpNode` must implement the `Comparable` interface, and that the objects of the actual type parameter must be comparable to each other. This implies that the type, say `A`, that we can use to parameterize the generic class, must implement the parameterized interface `Comparable<A>`.

If we base the implementation of the `CmpNode` class on the generic class `Node<E>`, we can write the declaration as follows:

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E> {
  // ...
}
```

The `extends` clause is used in two different ways: for the generic class `CmpNode` to extend the class `Node<E>`, and to constrain the type parameter `E` of the generic class `CmpNode` to the `Comparable<E>` interface. Although the type parameter `E` must implement the interface `Comparable<E>`, we do *not* use the keyword `implements` in a constraint. Neither can we use the `super` clause to constrain the type parameter of a generic class.

If we want `CmpNode`s to have a natural ordering based on the natural ordering of their data values, we can declare the generic class `CmpNode` as shown in **Example 11.8**:

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {
```

```
  // ...
}
```

Note how the `Comparable` interface is parameterized in the `implements` clause. The constraint `<E extends Comparable<E>>` specifies that the type parameter `E` is `Comparable`, and the clause `implements Comparable<CmpNode<E>>` specifies that the generic class `CmpNode` is `Comparable`.

Example 11.8 *Implementing the* `Comparable<E>` *Interface*

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {

  CmpNode(E data, CmpNode<E> next) { super(data, next); }

  @Override public int compareTo(CmpNode<E> node2) {
    return this.getData().compareTo(node2.getData());
  }
}
```

Here are some examples of how the generic class `CmpNode` in **Example 11.8** can be parameterized:

Click here to view code image

```
CmpNode<Integer> intCmpNode  = new CmpNode<>(2020, null);          // (1)
CmpNode<Number>  numCmpNode  = new CmpNode<Number>(2020, null);   // (2) Error!
CmpNode<Integer> intCmpNode2 = new CmpNode<>(2021, null);
int result = intCmpNode.compareTo(intCmpNode2);
```

The actual type parameter `Integer` at (1) implements `Comparable<Integer>`, but the actual type parameter `Number` at (2) is not `Comparable`. In the declaration `CmpNode<A>`, the compiler ensures that `A` implements `Comparable<A>`.

Multiple Bounds

A bounded type parameter can have *multiple bounds*, B_1 & B_2 & ... & B_n, which must be satisfied by the *actual* type parameter:

Click here to view code image

```
class CmpNode<E extends Number & Serializable> ...
```

An extra bound, the `Serializable` interface, has been added using the ampersand (`&`). The formal type parameter `E` is a subtype of *both* `Number` and `Serializable`, and represents both of these concrete types in the body of the generic class. The constraint above will only allow the generic type to be parameterized by an actual type parameter which is a subtype of *both* `Number` and `Serializable`.

We can add as many bounds as necessary. A type parameter `E` having multiple bounds is a *subtype* of all of the types denoted by the individual bounds. A bound can be a parameterized type, as in the following generic class header:

Click here to view code image

```
class CmpNode<E extends Comparable<E> & Serializable> ...
```

If the raw type of a bound is a (non-final) superclass of the bounded type parameter, it can only be specified as the first bound, and there can only be one such bound (as a subclass can only extend one immediate superclass). The raw type of an individual bound cannot be used with different type arguments, since a type parameter cannot be the subtype of more than one bound having the same raw type. In the class header below, whatever `E` is, it cannot be a subtype of two parameterizations of the same interface type (i.e., `Comparable`) at the same time:

Click here to view code image

```
class CmpNode<E extends Comparable<E> & Serializable & Comparable<String>> //Error
```

If the type parameter has a bound, methods of the bound can be invoked on instances of the type parameter in the generic class. Otherwise, only methods from the `Object` class can be invoked on instances of the type parameter. In the declaration of the generic class `Node<E>` in **Example 11.2**, **p. 568**, we cannot call any methods on instances of the type parameter except for those in the `Object` class because the type parameter is unbounded. Since the instances of the type parameter `E` are guaranteed to be `Comparable<E>` in the generic class `CmpNode`, we can call the method `com-pareTo()` of the `Comparable` interface on these instances.

11.7 Generic Methods and Constructors

We first look at how generic methods and constructors are declared, and then at how they can be called—both with and without explicit actual type parameters.

Declaring Generic Methods

A generic method (also called *polymorphic method*) is implemented like an ordinary method, except that one or more formal type parameters are specified immediately pre-

ceding the return type. In the case of a generic constructor, the formal parameters are specified before the class name in the constructor header. Much of what applies to generic methods in this regard also applies to generic constructors.

Example 11.9 *Declaring Generic Methods*

Click here to view code image

```
public class Utilities {

  // The key type and the array element type can be any type.
  static boolean containsV1(Object key, Object[] array) { // (1) Non-generic
                                                          //     version
    for (Object element : array)
      if (key.equals(element)) return true;
    return false;
  }

  // The key type and the array element type are the same.
  static <E> boolean containsV2(E key, E[] array) {       // (2) Generic version
    for (E element : array)
      if (key.equals(element)) return true;
    return false;
  }

  // The key type is a subtype of the array element type.
  static <K extends E, E> boolean containsV3(K key, E[] array) {  // (3)
    for (E element : array)
      if (key.equals(element)) return true;
    return false;
  }
}
```

In **Example 11.9**, the method `containsV1()` at (1) is a non-generic method to determine the membership of an arbitrary key in an arbitrary array of objects.

Click here to view code image

```
static boolean containsV1(Object key, Object[] array) { // (1) Non-generic version
  // ...
}
```

The method declaration at (1) is too general, in the sense that it does not express any relationship between the key and the array. This kind of type dependency between parameters can be achieved by using generic methods. In **Example 11.9**, the method `containsV2()` at (2) is a generic method to determine the membership of a key of type `E` in an array of type `E`. The type `Object` at (1) has been replaced by the type parameter `E`

at (2), with the formal type parameter `E` being specified before the return type, in the same way as for a generic type.

Click here to view code image

```
static <E> boolean containsV2(E key, E[] array) {       // (2) Generic version
  // ...
}
```

As with the generic types, a formal type parameter can have a bound, which is a type (i.e., not a type parameter). A formal type parameter can be used in the return type, in the formal parameter list, and in the method body. It can also be used to specify *bounds* in the formal type parameter list.

A generic method need not be declared in a generic type. If declared in a generic type, a *generic instance method* can also use the type parameters of the generic type as any other non-generic instance methods of the generic type. In contrast, a *generic static method* can only use the type parameters declared in its method header.

Calling Generic Methods

Consider the following class declaration:

Click here to view code image

```
public class ClassDecl {
  static <E_1,..., E_k> void genericMethod(P_1 p_1,..., P_m p_m) { ... }
  // ...
}
```

Note that in the method declaration above, a type `P_i` may or may not be from the list of type parameters `E_1`, ..., `E_k`. We can call the method in various ways. One main difference from calling a non-generic method is that the actual type parameters can be specified before the method name in the call to a generic method. In the method calls shown below, `<A_1,..., A_k>` are the actual type parameters and `(a_1,..., a_m)` are the actual arguments. The specification `<A_1,..., A_k>` of the actual type parameters is known as a *type witness*, as it corroborates the types to use in the method call. If included, it must be specified in its entirety. If there is not type witness, then the compiler *infers* the actual type parameters.

The following method calls can occur in any static or non-static context where the class `CallDecl` is accessible:

Click here to view code image

```
CallDecl ref;
ref.<A_1,..., A_k>genericMethod(a_1,..., a_m);
CallDecl.<A_1,..., A_k>genericMethod(a_1,..., a_m);
```

The following method calls can only occur in a non-static context of the class `CallDecl`:

Click here to view code image

```
this.<A_1,..., A_k>genericMethod(a_1,..., a_m);            // Non-static context
super.<A_1,..., A_k>genericMethod(a_1,..., a_m);           // Non-static context
CallDecl.super.<A_1,..., A_k>genericMethod(a_1,..., a_m);  // Non-static context
```

Another difference from calling non-generic methods is that, if the type witness is explicitly specified, the syntax of a generic *static* method call requires an explicit reference or the raw type. When the type witness is not explicitly specified, the syntax of a generic method call is similar to that of a non-generic method call.

Click here to view code image

```
<A_1,..., A_k>genericMethod(a_1,..., a_m);     // Compile-time error!
genericMethod(a_1,..., a_m);                   // Ok.
```

Here are some examples of calls to the `containsV2()` method at (2) in the class `Utilities` in **Example 11.9**, where the type witness is specified. We can see from the method signature and the method call signature that the method can be applied to the arguments at (1), (2), and (3), but not at (4). At (5), we must specify a reference or the class name because a type witness with the actual type parameter is specified.

Click here to view code image

```
Integer[] intArray = {10, 20, 30};

boolean f1 = Utilities.<Integer>containsV2(20, intArray);           // (1) true
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Integer, Integer[])

boolean f2 = Utilities.<Number>containsV2(30.5, intArray);          // (2) false
// E is Number.
// Method signature:      containsV2(Number, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f3 = Utilities.<Comparable<Integer>> containsV2(20, intArray); // (3) true
// E is Comparable<Integer>.
// Method signature:      containsV2(Comparable<Integer>, Comparable<Integer>[])
// Method call signature: containsV2(Integer,             Integer[])
```

```
boolean f4 = Utilities.<Integer>containsV2(30.5, intArray);         // (4) Error!
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Double,  Integer[])

// Requires explicit reference or raw type.
boolean f5 = <Integer>containsV2(20, intArray);              // (5) Syntax error!
```

Here are some examples of method calls where the compiler infers the actual type parameters from the method call. At (6), both the key and the element type are `Integer`, the compiler infers that the actual type parameter is `Integer`. At (7), where the key type is `Double` and the element type is `Integer`, the compiler infers the actual type parameter to be `Number`—that is, the first common supertype of `Double` and `Integer`. At (8), the compiler infers the actual type parameter to be `Serializable`— that is, the first common supertype of `String` and `Integer`. In all the cases below, the method is applicable to the arguments.

Click here to view code image

```
boolean f6 = Utilities.containsV2(20, intArray);                    // (6) true
// E is inferred to be Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(Integer, Integer[])

boolean f7 = Utilities.containsV2(30.5, intArray);                  // (7) false;
// E is inferred to be Number.
// Method signature:      containsV2(Number, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f8 = Utilities.containsV2("Hi", intArray);                  // (8) false;
// E is inferred to be Serializable.
// Method signature:      containsV2(Serializable, Serializable[])
// Method call signature: containsV2(String, Integer[])
```

At (8), if we had specified the actual type parameter explicitly to be `Integer`, the compiler would flag an error, as shown at (9), since the method signature is not applicable to the arguments:

Click here to view code image

```
boolean f9 = Utilities.<Integer>containsV2("Hi", intArray);         // (9) Error!
// E is Integer.
// Method signature:      containsV2(Integer, Integer[])
// Method call signature: containsV2(String, Integer[])
```

We can explicitly specify the key type to be a subtype of the element type by introducing a new formal parameter and a bound on the key type, as for the method `containsV3()` at (3) in **Example 11.9**:

Click here to view code image

```
static <K extends E, E> boolean containsV3(K key, E[] array) {
  // ...
}
```

The following calls at (10) and (11) illustrates inferring of actual type parameters from the method call when no type witness is specified. At (10), the compiler infers the `K` type parameter to be `Double` and the type parameter `E` to be `Number`—that is, the first common supertype of `Double` and the array element type `Integer`. The constraint is satisfied and the method signature is applicable to the arguments.

Click here to view code image

```
boolean f10 = Utilities.containsV3(30.5, intArray);                 // (10) false
// K is inferred to be Double. E is inferred to be Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Double, Number[])
// Method call signature: containsV2(Double, Integer[])

boolean f11 = Utilities.containsV3("Hi", intArray);                 // (11) false
// K is inferred to be String. E is inferred to be Serializable.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(String, Serializable[])
// Method call signature: containsV2(String, Integer[])
```

At (11), the compiler infers the `K` type parameter to be `String` and the type parameter `E` to be `Serializable`—that is, the first common supertype of `String` and the array element type `Integer`. The constraint is satisfied and the method signature is applicable to the arguments, as both `String` and `Integer` are `Serializable`.

The examples below illustrate how constraints come into play in method calls. At (12), the constraint is satisfied and the method signature is applicable to the arguments. At (13), the constraint is not satisfied; therefore, the call is rejected. At (14), the constraint is satisfied, but the method signature is not applicable to the arguments. The call at (15) is rejected because the number of actual type parameters specified in the call is incorrect.

Click here to view code image

```
boolean f12 = Utilities.<Number, Number>containsV3(30.0, intArray); // (12) false
// K is Number. E is Number.
// The constraint (K extends E) is satisfied.
```

```
// Method signature:      containsV3(Number, Number[])
// Method call signature: containsV3(Double, Integer[])

boolean f13 = Utilities.<Number, Integer>
                        containsV3(30.5, intArray);                 // (13) Error!
// K is Number. E is Integer.
// The constraint (K extends E) is not satisfied.

boolean f14 = Utilities.<Integer, Number>
                        containsV3(30.5, intArray);                 // (14) Error!
// K is Integer. E is Number.
// The constraint (K extends E) is satisfied.
// Method signature:      containsV3(Integer, Number[])
// Method call signature: containsV3(Double, Integer[])

boolean f15 = Utilities.<Number>containsV3(30.5, intArray);         // (15) Error!
// Incorrect no. of type parameters.
```

Typically, the dependencies among the parameters of a method and its return type are expressed by formal type parameters. Here are some examples:

Click here to view code image

```
public static <K,V> Map<V,List<K>> toMultiMap(Map<K,V> origMap) { ... } // (16)
public static <N> Set<N> findVerticesOnPath(Map<N,Collection<N>> graph,
                                            N startVertex)      { ... } // (17)
```

The method header at (16) expresses the dependency that the map returned by the method has the values of the original map as keys, and its values are lists of keys of the original map—that is, the method creates a *multimap*. In the method header at (17), the type parameter `N` specifies the element type of the set of vertices to be returned, the type of the keys in the map, the element type of the collections that are values of the map, and the type of the start vertex.

11.8 Implementing a Simplified Generic Stack

The `Node<E>` class from **Example 11.2**, **p. 568**, can be used to implement linked data structures. **Example 11.10** is an implementation of a simplified generic stack using the `Node<E>` class. The emphasis is not on how to develop a full-blown, industrial-strength implementation, but on how to present a simple example in the context of this book in order to become familiar with code that utilizes generics. For thread-safety issues concerning a stack, see **§22.4**, **p. 1396**.

The class `MyStack<E>` implements the interface `IStack<E>` shown in **Example 11.10**, and uses the class `Node<E>` from **Example 11.2**. The class `NodeIterator<E>` in **Example 11.10** provides an iterator to iterate over linked nodes. The class `MyStack<E>` is `Iterable<E>`,

meaning we can use the `for(:)` loop to iterate over a stack of this class (see (9) and (12)). It is instructive to study the code to see how type parameters are used in various contexts, how the iterator is implemented, and how we can use the `for(:)` loop to iterate over a stack. For details on the `Iterable<E>` and `Iterator<E>` interfaces, see §15.2, p. 791.

Example 11.10 *Implementing a Simplified Generic Stack*

Click here to view code image

```
/** Interface of a generic stack */
public interface IStack<E> extends Iterable<E> {
  void push(E element);           // Add the element to the top of the stack
  E pop();                        // Remove the element at the top of the stack.
  E peek();                       // Get the element at the top of the stack.
  int size();                     // No. of elements on the stack.
  boolean isEmpty();              // Determine if the stack is empty.
  boolean isMember(E element);    // Determine if the element is in the stack.
  E[] toArray(E[] toArray);       // Copy elements from stack to array
  @Override
  String toString();              // Return suitable text representation of
                                  // elements on the stack: (e1, e2, ..., en)
}
```

Click here to view code image

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/** Simplified implementation of a generic stack */
public class MyStack<E> implements IStack<E> {                    // (1)
  // Top of stack.
  private Node<E> tos;                                            // (2)
  // Size of stack
  private int numOfElements;                                      // (3)

  @Override public boolean isEmpty() { return tos == null; }      // (4)
  @Override public int size()        { return numOfElements; }    // (5)

  @Override public void push(E element) {                         // (6)
    tos = new Node<>(element, tos);
    ++numOfElements;
  }

  @Override public E pop() {                                      // (7)
    if (!isEmpty()) {
      E data = tos.getData();
      tos = tos.getNext();
      --numOfElements;
```

```
      return data;
    }
    throw new NoSuchElementException("No elements.");
  }

  @Override public E peek()  {                                        // (8)
    if (!isEmpty()) return tos.getData();
    throw new NoSuchElementException("No elements.");
  }

  // Membership
  @Override public boolean isMember(E element) {                      // (9)
    for (E data : this)
      if (data.equals(element))
        return true;       // Found.
    return false;          // Not found.
  }

  // Get iterator.
  @Override public Iterator<E> iterator() {                           // (10)
    return new NodeIterator<>(this.tos);
  }

  // Copy to array as many elements as possible.
  @Override public E[] toArray(E[] toArray) {                         // (11)
    Node<E> thisNode = tos;
    for (int i = 0; thisNode != null && i < toArray.length; i++) {
      toArray[i] = thisNode.getData();
      thisNode = thisNode.getNext();
    }
    return toArray;
  }

  // Text representation of stack: (e1, e2, ..., en).
  @Override public String toString() {                                // (12)
    StringBuilder rep = new StringBuilder("(");
    for (E data : this) {
      rep.append(data + ", ");
    }
    if (!isEmpty()) {
      int len = rep.length();
      rep.delete(len - 2, len);                   // Delete the last ", ".
    }
    rep.append(")");
    return rep.toString();
  }
}
```

Click here to view code image

```
import java.util.Iterator;

/** Iterator for nodes */
public class NodeIterator<E> implements Iterator<E> {
  private Node<E> thisNode;

  public NodeIterator(Node<E> first) { thisNode = first;  }

  @Override public boolean hasNext() { return thisNode != null; }

  @Override public E next() {
    E data = thisNode.getData();
    thisNode = thisNode.getNext();
    return data;
  }

  @Override public void remove() { throw new UnsupportedOperationException(); }
}
```

Review Questions

11.1 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_50 {
  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();     // (1)
    intList.add(2020);
    intList.add(2021);
    List<Number> numList = intList;                // (2)
    for (Number n : numList)                       // (3)
      System.out.println(n + " ");
  }
}
```

Select the one correct answer.

a. The code will fail to compile because of an error at (1).

b. The code will fail to compile because of an error at (2).

c. The code will fail to compile because of an error at (3).

d. The code will compile. When run, it will throw a `ClassCastException` at (3).

e. The code will compile and execute normally.

11.2 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_40 {
  public static void main(String[] args) {
    List <? super Integer> sList = new ArrayList<Number>(); //(1)
    int i = 2020;
    sList.add(i);
    sList.add(++i);                                          //(2)
    Number num = sList.get(0);                               //(3)
  }
}
```

11.3 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_70 {
  public static void main(String[] args) {
    List<Integer> glst1 = new ArrayList();       //(1)
    List nglst1 = glst1;                         //(2)
    List nglst2 = nglst1;                        //(3)
    List<Integer> glst2 = glst1;                 //(4)
  }
}
```

Select the one correct answer.

a. The code will compile without any warnings.

b. The code will compile with an unchecked warning at (1).

c. The code will compile with an unchecked warning at (2).

d. The code will compile with an unchecked warning at (3).

e. The code will compile with an unchecked warning at (4).

f. The code will fail to compile.

11.4 Which occurrences of the type parameter `T` are illegal in the following class?

Click here to view code image

```
public class Box<T> {
  private T item;                                        // (1)

  private static T[] storage = new T[100];               // (2)

  public Box(T item) { this.item = item; }               // (3)

  public T getItem() { return item; }                    // (4)

  public void setItem(T newItem) { item = newItem; }     // (5)

  public static void getAllItems(T newItem) {            // (6)
    T temp;                                              // (7)
  }
}
```

Select the three correct answers.

a. The occurrence of the type parameter `T` at (1)

b. The occurrence of the type parameter `T` at (2)

c. The occurrence of the type parameter `T` at (3)

d. The occurrence of the type parameter `T` at (4)

e. The occurrence of the type parameter `T` at (5)

f. The occurrence of the type parameter `T` at (6)

g. The occurrence of the type parameter `T` at (7)

11.5 Which of the following declarations will not result in compile-time errors? Select the four correct answers.

a.

Click here to view code image

```
Map<Integer, Map<Integer, String>> map1
              = new HashMap<Integer, HashMap<Integer, String>>();
```

b.

Click here to view code image

```
Map<Integer, HashMap<Integer, String>> map2 = new HashMap<>();
```

c.

Click here to view code image

```
Map<Integer, Integer> map3 = new HashMap<Integer, Integer>();
```

d.

Click here to view code image

```
Map<? super Integer, ? super Integer> map4
              = new HashMap<? super Integer, ? super Integer>();
```

e.

Click here to view code image

```
Map<? super Integer, ? super Integer> map5 = new HashMap<Number, Number>();
```

f.

Click here to view code image

```
Map<? extends Number, ? extends Number> map6
              = new HashMap<Number, Number>();
```

g.

Click here to view code image

```
Map <?,?> map7 = new HashMap<?,?>();
```

11.6 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
class Fruit {}
```

```
class Apple extends Fruit {}

public class RQ100_15 {
  public static void main(String[] args) {
    List<? extends Apple> lst1 = new ArrayList<Fruit>(); // (1)
    List<? extends Fruit> lst2 = new ArrayList<Apple>(); // (2)
    List<? super Apple> lst3 = new ArrayList<Fruit>();   // (3)
    List<? super Fruit> lst4 = new ArrayList<Apple>();   // (4)
    List<?> lst5 = lst1;                                 // (5)
    List<?> lst6 = lst3;                                 // (6)
    List lst7 = lst6;                                    // (7)
    List<?> lst8 = lst7;                                 // (8)
  }
}
```

Select the one correct answer.

a. (1) will compile, but (2) will not.

b. (3) will compile, but (4) will not.

c. (5) will compile, but (6) will not.

d. (7) will compile, but (8) will not.

e. None of the above

11.7 Which statement is true about the following program?

Click here to view code image

```
import java.util.*;
public class RQ100_11 {
  public static void main(String[] args) {
    Set set = new TreeSet<String>();
    set.add("one");
    set.add(2);
    set.add("three");
    System.out.println(set);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile with unchecked warnings, and will print the elements in the set.

c. The program will compile without unchecked warnings, and will print the elements in the set.

d. The program will compile with unchecked warnings, and will throw an exception at runtime.

e. The program will compile without unchecked warnings, and will throw an exception at runtime.

11.8 Which of the following statements are true about the following program?

Click here to view code image

```
class Vehicle {}
class Car extends Vehicle {}
class Sedan extends Car {}

class Garage<V> {
  private V v;
  public V get() { return this.v; }
  public void put(V v) { this.v = v; }
}

public class GarageAdmin {

  private Object object = new Object();
  private Vehicle vehicle = new Vehicle();
  private Car car = new Car();
  private Sedan sedan = new Sedan();

  public void doA(Garage g) {
    g.put(object);            // (1)
    g.put(vehicle);           // (2)
    g.put(car);               // (3)
    g.put(sedan);             // (4)
    object  = g.get();        // (5)
    vehicle = g.get();        // (6)
    car     = g.get();        // (7)
    sedan   = g.get();        // (8)
  }
}
```

Select the two correct answers.

a. The call to the `put()` method in statements (1) through (4) will compile.

b. The assignment statement (5) will compile.

c. The assignment statements (6), (7), and (8) will compile.

11.9 Which method calls can be inserted individually at (1) so that the program will compile without warnings?

Click here to view code image

```
import java.util.*;
public class GenParam {
  public static void main(String[] args) {
    List<Number> numList = new ArrayList<>();
    List<Integer> intList = new ArrayList<>();
    // (1) INSERT CODE HERE
  }

  static <T> void move(List<? extends T> lst1, List<? super T> lst2) {}
}
```

Select the three correct answers.

a. `GenParam.move(numList, intList);`

b. `GenParam.<Number>move(numList, intList);`

c. `GenParam.<Integer>move(numList, intList);`

d. `GenParam.move(intList, numList);`

e. `GenParam.<Number>move(intList, numList);`

f. `GenParam.<Integer>move(intList, numList);`

11.10 Given the class below, which declaration statement is not valid?

Click here to view code image

```
class Box<T> {
      Box()         {System.out.println(this);}              // (1)
  <V> Box(V v)      {System.out.println(v);}                  // (2)
  <V> Box(T t, V v) {System.out.println(t + ", " + v);}      // (3)
}
```

Select the one correct answer.

a. `Box<String> ref1 = new Box<>();`

b. `Box<String> ref2 = new Box<>("one");`

c. `Box<String> ref3 = new Box<>(2020);`

d. `Box<Integer> ref4 = new Box<>(2020, "one");`

e. `Box<String> ref5 = new Box<>("one", 2020);`

f. `Box<Integer> ref6 = new Box<>("one", 2020);`

11.9 Wildcard Capture

As we have seen, a wildcard can represent a family of types. However, the compiler needs to have a more concrete notion of a type than a wildcard in order to do the necessary type checking. Internally, the compiler represents the wildcard by some anonymous but specific type. Although this type is unknown, it belongs to the family of types represented by the wildcard. This specific but unknown type is called the *capture of* the wildcard.

Compiler messages about erroneous usage of wildcards often refer to the capture of a wildcard. Here are some examples of such error messages, based on compiling the following code:

Click here to view code image

```
// File: WildcardCapture.java
...
Node<?>                 anyNode;
Node<? super Number>    supNumNode;

Node<Integer> intNode = anyNode;                // (1) Compile-time error!
Node<? extends Number> extNumNode = supNumNode; // (2) Compile-time error!
anyNode.setData("Trash");                       // (3) Compile-time error!
```

The assignment at (1) results in the following rather cryptic error message:

Click here to view code image

```
WildcardCapture.java:10: error: incompatible types: Node<CAP#1> cannot be
converted to Node<Integer>
    Node<Integer> intNode = anyNode;                // (1) Compile-time error!
                            ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

The type of the reference `anyNode` is `Node<CAP#1>`. The name `CAP#1` is used by the compiler to designate the *type capture* of the wildcard (`"capture of ?"`) at (1). The type of the

reference `intNode` is `Node<Integer>`. The reference value of a `Node<CAP#1>` cannot be assigned to a `Node<Integer>` reference. Whatever the type capture of the wildcard is, it cannot be guaranteed to be `Integer`, and the assignment is rejected. To put it another way, the assignment involves a narrowing reference conversion, requiring an explicit cast which is not provided: `Node<?>` is the supertype of all invocations of the generic class `Node<E>`.

The error message below for the assignment at (2) shows the type capture `CAP#1` of the lower bounded wildcard at (2) to be `"capture of ? super Number"`. **Figure 11.5**, **p. 583**, also shows that the `Node<capture of ? super Number>` and `Node<? extends Number>` types are unrelated.

Click here to view code image

```
WildcardCapture.java:11: error: incompatible types: Node<CAP#1> cannot be
converted to Node<? extends Number>
    Node<? extends Number> extNumNode = supNumNode; // (2) Compile-time error!
                                        ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object super: Number from capture of ? super Number
```

The method call at (3) results in the following error message:

Click here to view code image

```
WildcardCapture.java:12: error: incompatible types: String cannot be converted to
CAP#1
    anyNode.setData("Trash");                       // (3) Compile-time error!
                    ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

The type of the reference `anyNode` is `Node<?>` and the type of the formal parameter in the method declaration is `CAP#1`, where `CAP#1` is `"capture of ?"`. The type of the actual parameter in the method call is `String`, which is not compatible with `CAP#1`. The call is not allowed. As we have seen earlier, with a `<?>` reference we cannot put anything into a data structure, except `null` s.

If we have the following method in the class `MyStack`:

Click here to view code image

```
public static <T> void move(MyStack<? extends T> srcStack,
                            MyStack<? super T> dstStack) {
  while (!srcStack.isEmpty())
```

```
    dstStack.push(srcStack.pop());
}
```

and we try to compile the following client code in the source file `MyStackUser.java`:

Click here to view code image

```
MyStack<?> anyStack;
MyStack.move(anyStack, anyStack);   // Compile-time error!
```

the compiler issues the following error message:

Click here to view code image

```
MyStackUser.java:68: error: method move in class MyStack<E#2> cannot be applied to
given types;
      MyStack.move(anyStack, anyStack);     // Compile-time error!
             ^
  required: MyStack<? extends E#1>,MyStack<? super E#1>
  found: MyStack<CAP#1>,MyStack<CAP#2>
  reason: cannot infer type-variable(s) E#1
    (argument mismatch; MyStack<CAP#2> cannot be converted to MyStack<? super E#1>)
  where E#1,E#2 are type-variables:
    E#1 extends Object declared in method <E#1>move(MyStack<? extends E#1>,My-
Stack<? super E#1>)
    E#2 extends Object declared in class MyStack
  where CAP#1,CAP#2 are fresh type-variables:
    CAP#1 extends Object from capture of ?
    CAP#2 extends Object from capture of ?
```

The error message shows that each occurrence of a wildcard in a statement is represented by a distinct type capture, namely `CAP#1` and `CAP#2`. We see that the signature of the `move()` method is `move(MyStack<? extends E#1>, MyStack<? super E#1>)`. The type of the reference `anyStack` is `MyStack<?>`. The static types of the two arguments in the method call are `MyStack<CAP#1>` and `MyStack<CAP#2>`, where `CAP#1` and `CAP#1` designate two distinct type captures (`capture of ?`). The signature of the argument list is (`MyStack<CAP#1>`, `MyStack<CAP#2>`). The type parameter `T` cannot be inferred from the types of the arguments, as the stacks are considered to be of two different types, and therefore, the call is rejected.

Capture Conversion

Consider the following non-generic method which does not compile:

Click here to view code image

```
static void fillWithFirstV1(List<?> list) {
  Object firstElement = list.get(0);       // (1)
  for (int i = 1; i < list.size(); i++)
    list.set(i, firstElement);             // (2) Compile-time error
}
```

The method should fill any list passed as an argument with the element in its first position. The call to the `set()` method at (2) is not permitted, as a *set* operation is not possible with a `<?>` reference (see **Table 11.3**, **p. 590**). Using the unbounded wildcard `?` to parameterize the list does not work. We can replace the wildcard with a type parameter of a *generic method*, as follows:

Click here to view code image

```
static <E> void fillWithFirstV2(List<E> list) {
    E firstElement = list.get(0);          // (3)
    for (int i = 1; i < list.size(); i++)
      list.set(i, firstElement);           // (4)
  }
```

Since the type of the argument is `List<E>`, we can set and get objects of type `E` from the list. We have also changed the type of the reference `firstElement` from `Object` to `E` in order to set the first element in the list.

It turns out that if the first method `fillWithFirstV1()` is reimplemented with a call to the generic method `fillWithFirstV2()`, it all works well:

Click here to view code image

```
static void fillWithFirstV3(List<?> list) {
  fillWithFirstV2(list);                   // (5) Type conversion
}
```

The wildcard in the argument of the `fillWithFirstV3()` method has a type capture. In the call to the `fillWithFirstV2()` method at (5), this type capture is converted to the type `E`. This conversion is called *capture conversion*, and it comes into play under certain conditions, which are beyond the scope of this book.

11.10 Flexibility with Wildcard Parameterized Types

Nested Wildcards

In this subsection, the examples make use of type parameters that are specified for the interfaces `Collection<E>`, `Set<E>`, and `Map<K,V>` in the j`ava.util` package (**Chapter 15**,

p. 781). In the discussion below, the important fact to keep in mind is that the interface `Set<E>` is a subinterface of `Collection<E>`.

We have seen that the subtype relationship is invariant for the unbounded type parameter `<T>`:

Click here to view code image

```
Collection<Number> colNum;
Set<Number> setNum;
Set<Integer> setInt;
colNum = setNum; // (1) Set<Number> <: Collection<Number>
colNum = setInt; // (2) Compile-time error!
```

The same is true when concrete parameterized types are used as actual type parameters, implementing what are called *nested parameterized types*—that is, using parameterized types as type parameters.

Click here to view code image

```
Collection<Collection<Number>> colColNum; // Collection of Collections of Number
Set<Collection<Number>>        setColNum; // Set of Collections of Number
Set<Set<Integer>>              setSetInt; // Set of Sets of Integer
colColNum = setColNum;                    // (3) Set<Collection<Number>> <:
                                          //     Collection<Collection<Number>>
colColNum = setSetInt;                    // (4) Compile-time error!
setColNum = setSetInt;                    // (5) Compile-time error!
```

Again, we can use the upper bounded wildcard to induce subtype covariance. The upper bounded wildcard is applied at the *top level* in the code below. The assignment below at (8) is not compatible because `Set<Set<Integer>>` is *not* a subtype of

Click here to view code image

```
Collection<? extends Collection<Number>> colExtColNum;
colExtColNum = colColNum;       // (6) Collection<Collection<Number>> <:
                                //     Collection<? extends Collection<Number>>
colExtColNum = setColNum;       // (7) Set<Collection<Number>> <:
                                //     Collection<? extends Collection<Number>>
colExtColNum = setSetInt;       // (8) Compile-time error!
```

In the code below, the wildcard is applied at the *innermost* level:

Click here to view code image

```
Collection<Collection<? extends Number>> colColExtNum;
colColExtNum = colColNum;       // (9)  Compile-time error!
colColExtNum = setColNum;       // (10) Compile-time error!
colColExtNum = setSetInt;       // (11) Compile-time error!
```

The assignments above show that the upper bounded wildcard induces subtype covariance *only* at the top level. At (9), type `A` (= `Collection<Number>`) is a subtype of type `B` (= `Collection<? extends Number>`), but because a subtype covariance relationship does not hold between parameterized types, the type `Collection<A>` (= `Collection<Collection<Number>>`) is *not* a subtype of `Collection<B>` (= `Collection<Collection<? extends Number>>`).

The above discussion also applies when a parameterized type has more than one type parameter:

Click here to view code image

```
Map<Number, String>  mapNumStr;
Map<Integer, String> mapIntStr;
mapNumStr = mapIntStr;             // (12) Compile-time error!
```

Again, the upper bounded wildcard can only be used at the top level to induce subtype covariance:

Click here to view code image

```
Map<Integer, ? extends Collection<String>> mapIntExtColStr;
Map<Integer, Collection<? extends String>> mapIntColExtStr;
Map<Integer, Collection<String>>           mapIntColStr;
Map<Integer, Set<String>>                  mapIntSetStr;
mapIntExtColStr = mapIntColStr;// (13) Map<Integer, Collection<String>> <:
                               //      Map<Integer, ? extends Collection<String>>
mapIntExtColStr = mapIntSetStr;// (14) Map<Integer, Set<String>> <:
                               //      Map<Integer, ? extends Collection<String>>
mapIntColStr    = mapIntSetStr;    // (15) Compile-time error!
mapIntColExtStr = mapIntColStr;    // (16) Compile-time error!
mapIntColExtStr = mapIntSetStr;    // (17) Compile-time error!
```

Wildcard Parameterized Types as Formal Parameters

We now examine the implications of using wildcard parameterized types to declare formal parameters of a method.

We want to add a method in the class `MyStack<E>` (**Example 11.10**, **p. 598**) for moving the elements of a source stack to the current stack. Here are three attempts at implementing

such a method for the class `MyStack<E>`:

Click here to view code image

```
public void moveFromV1(MyStack<E> srcStack) {            // (1)
  while (!srcStack.isEmpty())
    this.push(srcStack.pop());
}
public void moveFromV2(MyStack<? extends E> srcStack) {  // (2)
  while (!srcStack.isEmpty())
    this.push(srcStack.pop());
}
public void moveFromV3(MyStack<? super E> srcStack) {    // (3)
  while (!srcStack.isEmpty())
    this.push(srcStack.pop());                           // Compile-time error!
}
```

Given the following three stacks:

Click here to view code image

```
MyStack<Number> numStack   = new MyStack<>();                  // Stack of Number
numStack.push(5.5); numStack.push(10.5); numStack.push(20.5);
MyStack<Integer> intStack1 = new MyStack<>();                  // Stack of Integer
intStack1.push(5); intStack1.push(10); intStack1.push(20);
MyStack<Integer> intStack2 = new MyStack<>();                  // Stack of Integer
intStack2.push(15); intStack2.push(25); intStack2.push(35);
```

We can only move elements between stacks of the same concrete type with the method at (1). The compile-time error below is due to the fact that `MyStack<Integer>` is not a subtype of `MyStack<Number>`.

Click here to view code image

```
intStack1.moveFromV1(intStack2); // Ok.
numStack.moveFromV1(intStack2);  // Compile-time error!
```

We can also move elements from a stack of type `MyStack<? extends E>` to the current stack, using the method at (2). This is possible because a reference of a type `MyStack<? extends E>` can refer to a stack with objects of type `E` or its subclass, and the *get* operation (i.e., the `pop()` method) is permissible, returning an object which has an actual type bounded by the upper bound `E`. The returned object can always be put into a stack of type `E` or its supertype.

Click here to view code image

```
intStack1.moveFromV2(intStack2);  // Pop from intStack2. Push on intStack1.
numStack.moveFromV2(intStack2);   // Pop from intStack2. Push on numStack.
```

The method at (3) will only allow `Object`s to be popped from a stack of type `MyStack<? super E>`, which could only be pushed onto a stack of type `Object`. Since `E` cannot be determined at compile time, the `push()` operation on the current stack is not permitted. Of the first two methods, the method at (2) is more flexible in permitting the widest range of calls for copying from the source stack to the current stack.

Similarly, we can add a method in the class `MyStack<E>` for moving the elements of the current stack to a destination stack:

Click here to view code image

```
public void moveToV1(MyStack<E> dstStack) {               // (4)
  while (!this.isEmpty())
    dstStack.push(this.pop());
}

public void moveToV2(MyStack<? extends E> dstStack) {     // (5)
  while (!this.isEmpty())
    dstStack.push(this.pop());                            // Compile-time error!
}

public void moveToV3(MyStack<? super E> dstStack) {       // (6)
  while (!this.isEmpty())
    dstStack.push(this.pop());
}
```

In the method at (5), the reference of type `MyStack<? extends E>` does not allow any *set* operations (in this case, the `push()` method) on the destination stack. The method at (6) provides the most flexible solution, as a reference of type `MyStack<? super E>` permits *set* operations for objects of type `E` or its subtypes:

Click here to view code image

```
intStack1.moveToV1(intStack2);    // Pop from intStack1. Push on intStack2.
intStack1.moveToV1(numStack);     // Compile-time error!

intStack1.moveToV3(intStack2);    // Pop from intStack1. Push on intStack2.
intStack1.moveToV3(numStack);     // Pop from intStack1. Push on numStack.
```

Evidently, the method at (6) is more flexible in permitting the widest range of calls for copying from the current stack to the destination stack.

Based on the discussion above, we can write a *generic method* for moving elements from a source stack to a destination stack. The following method signature is preferable, where objects of type `E` or its subtypes can be popped from the source stack and pushed onto a destination stack of type `E` or its supertype:

Click here to view code image

```
public static <E> void move(MyStack<? extends E> srcStack,  // (7)
                            MyStack<? super E> dstStack) {
  while (!srcStack.isEmpty())
    dstStack.push(srcStack.pop());
}

// Client code
MyStack.move(intStack1, intStack2);
MyStack.move(intStack1, numStack);
MyStack.move(numStack, intStack2);     // Compile-time error!
```

It is a common idiom to use wildcards as shown above in the method at (7), as the upper bounded wildcard (`? extends Type`) can be used to *get* objects from a data structure, and the lower bounded wildcard (`? super Type`) can be used to *set* objects in a data structure. Using wildcards in the method signature can increase the utility of a method, especially when explicit type parameters are specified in the method call.

Recursive Type Bounds Revisited

The class `MonoNode` and the interface `IMonoLink<E>` are declared in **Example 11.3**, **p. 572**, and the class `BiNode` and the interface `IBilink<E>` are declared in **Example 11.5**, **p. 574**. See also **Figure 11.1**, **p. 574**.

Click here to view code image

```
class MonoNode<E> implements IMonoLink<E> {
  private E                data;    // Data
  private IMonoLink<E>     next;    // Reference to next node       (1)
  // ...
}

class BiNode<E> extends MonoNode<E> implements IBiLink<E> {
  private IBiLink<E> previous;      // Reference to previous node   (2)
  // ...
}
```

Note that the `next` field has the type `IMonoLink<E>`, but the `previous` field has the type `IBiLink<E>`, and that the type `IBiLink<E>` is a subtype of the type `IMonoLink<E>`. This means that when iterating over a linked structure constructed from nodes that implement

the `IBiLink<E>` interface, we have to be careful. The method `traverseBinTree()` below traverses a binary tree constructed from such nodes. The method prints the data in the nodes. Note that it is necessary to cast the reference value returned by the `getNext()` method, as shown at (3), from the supertype `IMonoLink<E>` to the subtype `IBiLink<E>`.

Click here to view code image

```
public static <E> void traverseBinTree(IBiLink<E> root) {
  if (root.getPrevious() != null)
    traverseBinTree(root.getPrevious());
  System.out.print(root.getData() + ", ");
  if (root.getNext() != null)
    traverseBinTree((IBiLink<E>)root.getNext());  // (3) Cast necessary.
}
```

Example 11.11 declares a class called `RecNode`. The header of this class at (1) is declared as follows:

Click here to view code image

```
abstract class RecNode<E, T extends RecNode<E, T>>
```

The class specifies two type parameters: `E` and `T`. The type parameter `E` stands for the type of the data in the node, (2). The type parameter `T` stands for the type of the `next` field in the node, (3). It has the upper bound `RecNode<E, T>`, meaning that `T` must be a subtype of the bound—that is, of `RecNode<E, T>`. In other words, the class `RecNode` can only be parameterized by itself or its subclasses. The two parameters `E` and `T` are used in the class for their respective purposes.

Example 11.11 *Using Recursive Bounds*

Click here to view code image

```
abstract class RecNode<E, T extends RecNode<E, T>> {      // (1)
  private E data;                                         // (2)
  private T next;                                         // (3)

  RecNode(E data, T next) {
    this.data = data;
    this.next = next;
  }
  public void setData(E obj)  { data = obj; }
  public E    getData()       { return data; }
  public void setNext(T next) { this.next = next; }
  public T    getNext()       { return next; }
  @Override public String toString() {
```

```
    return this.data + (this.next == null ? "" : ", " + this.next);
  }
}
```

Click here to view code image

```
final class RecBiNode<E> extends RecNode<E, RecBiNode<E>> {          // (4)

  private RecBiNode<E>  previous;    // Reference to previous node   // (5)

  RecBiNode(E data, RecBiNode<E> next, RecBiNode<E> previous) {
    super(data, next);
    this.previous = previous;
  }
  public void setPrevious(RecBiNode<E> previous) { this.previous = previous; }
  public RecBiNode<E> getPrevious()              { return this.previous; }
  @Override public String toString() {
    return (this.previous == null? "" : this.previous + ", ") +
        this.getData() + (this.getNext() == null? "" : ", " + this.getNext());
  }
}
```

Example 11.11 declares another class, called `RecBiNode`. The header of this class at (4) is declared as follows:

Click here to view code image

```
final class RecBiNode<E> extends RecNode<E, RecBiNode<E>>
```

Note that the class has only one type parameter, `E`, that represents the data type in the node. The class extends the `RecNode` class, which is parameterized with the data type `E` and the type `RecBiNode<E>` to represent the type of the `next` field in the superclass `RecNode`. The class `RecBiNode` also declares a `previous` field of type `RecBiNode<E>` at (5), and the corresponding getter and setter methods. The upshot of this class declaration is that, for a node of type `RecBiNode<E>`, both the `next` and the `previous` fields have the same type as a node of this class. The traversal method can now be written without using any casts, passing it a node of the subtype `RecBiNode<E>`:

Click here to view code image

```
  public static <E> void traverseBinTree(RecBiNode<E> root) {     // (2)
    if (root.getPrevious() != null)
      traverseBinTree(root.getPrevious());
    System.out.print(root.getData() + ", ");
    if (root.getNext() != null)
```

```
        traverseBinTree(root.getNext());              // No cast necessary!
    }
```

The class declaration at (1) in **Example 11.11** uses what is called a *recursive type bound* in its constraint `T extends RecNode<E,T>`. New subtypes of the class `RecNode` can be implemented using this idiom, and the type of the `next` field will be the same as the subtype, as in the case of the subtype `RecBiNode<E>`. Recursive type bounds allow subclasses to use their own type, avoiding explicit casts to convert from the superclass.

Earlier in this chapter we saw an example of a recursive type bound defined by the constraint `T extends Comparable<T>`. Another example of a recursive type bound is the declaration of the `Enum<E extends Enum<E>>` class in the Java standard library.

11.11 Type Erasure

Understanding translation by type erasure aids in understanding the restrictions and limitations that arise when using generics in Java. Although the compiler generates generic-free bytecode, we can view the process as a source-to-source translation that generates non-generic code from generic code.

The translated code has no information about type parameter. That is, the type parameters have been *erased*—hence the term *type erasure*. This involves replacing the usage of the type parameters with concrete types, and inserting suitable type conversions to ensure type correctness. In certain situations, *bridge methods* are also inserted for backward compatibility.

Translation by Type Erasure

The process of determining the *erasure* of a type—that is, what a type in the source code should be replaced with—uses the following rules:

1. Drop all type parameter specifications from parameterized types.
2. Replace any type parameter as follows:

a. Replace it with the erasure of its bound, if it has one.

b. Replace it with `Object`, if it has none.

c. Replace it with the erasure of the *first* bound, if it has multiple bounds.

Table 11.4 shows examples of translation by erasure for some representative types, and the rules that are applied.

Table 11.4 *Examples of Type Erasure*

Type	Erasure	Rule no.

`List<E>`
`List<Integer>`
`List<String>`
`List<List<String>>`
`List<? super Integer>`
`List<? extends Number>`
	`List`	1

`List<Integer>[]`	`List[]`	1
`List`	`List`	1
`int`	`int`	For any primitive type
`Integer`	`Integer`	For any non-generic type

`class Subclass`
`  extends Superclass`
`  implements`
`  Comparable<Subclass> {...}`
	`class Subclass`
	`  extends Superclass`
	`  implements`
	`  Comparable {...}`
		1

`public static`
`  <T extends Comparable<? super T>>`
`  T`
`  max(T obj1,`
`  T obj2)`
`  { ... }`
	`public static`
	`  Comparable`
	`  max(Comparable obj1,`
	`  Comparable obj2)`
	`  { ... }`
		2a. The first bound is `Comparable`.

`public static <T> T`
`  doIt(T t)`
`  { T lv = t; }`
	`public static Object`
	`  doIt(Object t)`
	`  { Object lv = t; }`
		2b

`T extends MyClass & Comparable<T> & Serializable`	`MyClass`	2c. The first bound is `MyClass`.

The following code mixes legacy and generic code. Note that a `ClassCastException` is expected at (5) because the type-safety of the stack of `String` has been compromised.

Click here to view code image

```
// Pre-erasure code
List<String> strList = new ArrayList<>(); // (0)
List list = strList;       // (1) Assignment to non-generic reference is ok.
strList = list;            // (2) warning: unchecked conversion
strList.add("aha");        // (3) Method call type-safe.
list.add(23);              // (4) warning: [unchecked] unchecked call to add(E)
                           //     as a member of the raw type java.util.List
System.out.println(strList.get(1).length());  // (5) ClassCastException
```

It is instructive to compare the corresponding lines of code in the pre-erasure code above and the post-erasure results shown below. A cast is inserted to convert from `Object` type to `String` type at (5'). This is necessary because post-erasure code can only get an `Object` from the list, and in order to call the `length()` method, the reference value of this object must be converted to `String`. It is this cast that is the cause of the exception at runtime.

Click here to view code image

```
// Post-erasure code
List strList = new ArrayList();                          // (0')
List list = strList;                                     // (1')
strList = list;                                          // (2')
strList.add("aha");                                      // (3')
list.add(Integer.valueOf(23));                           // (4')
System.out.println(((String)strList.get(1)).length()); // (5') Cast inserted.
```

Bridge Methods

Bridge methods are inserted in subclasses by the compiler to ensure that overriding of methods works correctly. The canonical example is the implementation of the `Comparable` interface. The post-erasure code of the class `CmpNode<E>` from **Example 11.8** is shown below. A second `compareTo()` method has been inserted by the compiler at (2), whose method signature is `compareTo(Object)`. This is necessary because, without this method, the class would not implement the `Comparable` interface, as the post-erasure

`compareTo(Object)` method of the interface would not be overridden correctly in the erasure code.

Click here to view code image

```
class CmpNode extends Node implements Comparable {
  CmpNode(Object data, CmpNode next) {
    super(data, next);
  }
  public int compareTo(CmpNode node2) {       // (1) Implemented method erasure
    return this.getData().compareTo(node2.getData());
  }
  public int compareTo(Object node2) {        // (2) Inserted bridge method.
    return this.compareTo((CmpNode)node2);    // Calls the method at (1).
  }
}
```

11.12 Implications for Overloading and Overriding

Before discussing the implications generics have for overloading and overriding, we need a few definitions regarding method signatures.

Method Signatures Revisited

Method signatures play a crucial role in overloading and overriding of methods. The method signature comprises the method name and the formal parameter list. Two methods (or constructors) have the *same signature* if both of the following conditions are fulfilled:

- They have the *same name.*
- They have the *same formal parameter types.*

Two methods (or constructors) have the *same formal parameter types* if both of the following conditions are fulfilled:

- They have the *same number* of *formal parameters* and *type parameters*.
- The formal parameters and the bounds of the type parameters are the same after the occurrences of formal parameters in one are substituted with the corresponding types from the second.

The signature of a method `m()` is a *subsignature* of the signature of another method `n()`, if either one of these two conditions hold:

- Method `n()` has the *same* signature as method `m()`, or
- The signature of method `m()` is the same as the *erasure* of the signature of method `n()`.

The signatures of the two methods `m()` and `n()` are *override-equivalent* if either one of these two conditions hold:

- The signature of method `m()` is a *subsignature* of the signature of method `n()`, or
- The signature of method `n()` is a *subsignature* of the signature of method `m()`.

Implications for Overloading

Given the definitions above, we can now state that two methods are *overloaded* if they have the same name, but their signatures are *not override-equivalent*. Given the following three generic method declarations in a class:

Click here to view code image

```
static <T> void merge(MyStack<T> s1, MyStack<T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? extends T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? super T> s2) { /*...*/ }
```

After erasure, the signature of all three methods is:

```
merge(MyStack, MyStack)
```

That is, the signatures of the methods are override-equivalent, hence these methods are *not* overloaded. A class cannot contain two methods with override-equivalent signatures, and the compiler will report an error.

These three methods:

Click here to view code image

```
static <T> void merge(Node<T> s1, MyStack<T> s2)              { /*...*/ }
static <T> void merge(MyStack<T> s1, MyStack<? extends T> s2) { /*...*/ }
static <T> void merge(MyStack<T> s1, Node<? super T> s2)      { /*...*/ }
```

have the following signatures after erasure, respectively:

```
merge(Node, MyStack)
merge(MyStack, MyStack)
merge(MyStack, Node)
```

We can see that no two signatures are override-equivalent. Therefore, the three methods are overloaded.

The declaration of the class `MethodSGN` below shows some variations on the method signature. The resulting signature of the method header (which includes the method signature) is shown after erasure in the comment corresponding to the method.

Click here to view code image

```
class MethodSGN<T> {
  void doIt(boolean b) {}                          // (1) void doIt(boolean)

  void doIt(T t) {}                                // (2) void doIt(Object)

  List<StringBuilder> doIt(StringBuilder sb) {     // (3) List doIt(StringBuilder)
    return null;
  }

  <E extends Comparable<E>> void doIt(E element) // (4) void doIt(Comparable)
    {}
  <E> E doIt(MyStack<? extends E> stack) {         // (5) Object doIt(MyStack)
    return null;
  }
}
```

Adding any of the method declarations given below to the class `MethodSGN` would be an error, as each one of these method declarations has a method signature that is the same as one of the methods already in the class—that is, the signatures are override-equivalent.

Click here to view code image

```
void doIt(Object obj) {}                           // (2') void doIt(Object)

<E extends StringBuilder> List<E> doIt(E sb) {     // (3') List doIt(StringBuilder)
  return null;
}

void doIt(Comparable<T> element) {}                // (4') void doIt(Comparable)

<E> E doIt(MyStack<? super E> stack) {             // (5') Object doIt(MyStack)
  return null;
}
```

Implications for Overriding

The following conditions (referred to as *override criteria*) should be satisfied in order for a subtype method to override a supertype method:

- The signature of the subtype method is a *subsignature* of the signature of the supertype method (which is discussed in this subsection).

- Their *return types* should be compatible.
- Their `throws` *clauses* should be compatible (§7.5, p. 388).

Here we discuss the implication of method signatures for overriding.

The `@Override` *Annotation*

We can solicit the aid of the compiler to ensure that a method declaration in a subtype correctly overrides a method from its supertype. If a method declaration is preceded by the annotation `@Override` (§25.5, p. 1578), the compiler will issue an error if the method in the subtype does not override a method from its supertype. The examples in this book make heavy use of this annotation.

Example 11.12 illustrates the use of this annotation. The intention in the class `CmpNode` is to override the `equals()` method from the `Object` class and the `compareTo()` method from the `Comparable` interface. The error messages alert us to the fact that the annotated methods do not override any methods. The method signatures are not subsignatures of any methods that are inherited. The formal parameters are not correct for overriding at (1) and (2). Correct method headers are shown at (1') and (2').

Example 11.12 *Using the* `@Override` *Annotation*

Click here to view code image

```
class CmpNode<E extends Comparable<E>> extends Node<E>
                                       implements Comparable<CmpNode<E>> {

  CmpNode(E data, CmpNode<E> next) { super(data, next); }

  @Override
  public boolean equals(CmpNode node2) {           // (1) Compile-time error.
//public boolean equals(Object node2) {            // (1') Correct header.
    return this.compareTo(node2) == 0;
  }

  @Override
  public int compareTo(Object node2) {              // (2) Compile-time error.
//public int compareTo(CmpNode<E> node2) {          // (2') Correct header
    return this.getData().compareTo(node2.getData());
  }
}
```

Compiling the class `CmpNode`:

Click here to view code image

```
>javac CmpNode.java
...
CmpNode.java:8: method does not override or implement a method from a supertype
  @Override
  ^
...
CmpNode.java:14: method does not override or implement a method from a supertype
  @Override
  ^
```

Overriding Methods from Non-Generic Supertype

In **Example 11.13**, the signature at (1') is the same as the signature at (1): `set(Integer)`. The signature at (2') is the same as the *erasure* of the signature at (2): `set(List)`. The method at (2') shows a non-generic subtype method overriding a supertype method that uses generics. This is needed for legacy code: *Legacy supertypes* can be generified without it having consequences for any subtypes, as the signature of a subtype method that overrides a supertype method will be the same as the erasure of the signature of this supertype method.

Example 11.13 *Subsignatures*

Click here to view code image

```
import java.util.List;

class SupA {
  public void set(Integer ref)        {/*...*/}    // (1)
  public void set(List<Integer> list) {/*...*/}    // (2)
}

class SubA extends SupA {
  @Override public void set(Integer ref) {/*...*/}  // (1') same as at (1)
  @Override public void set(List list) {/*...*/}// (2') same as the erasure at (2)
}
```

Overriding Methods from Generic Supertype

In **Example 11.14**, both the subclasses `SubB1` and `SubB2` are subtypes of the parameterized supertype `SupB<Number>`—that is, `T` is `Number` in `SupB<T>`. At compile time, the signatures of the methods in `SubB1` are the same as the signatures of the methods in `SupB`; therefore, the methods are overridden. After erasure, the methods in `SupB` are equivalent to:

Click here to view code image

```
public void set(Object t) {/*...*/}              // (1)
public Object get()       {return null;}         // (2)
```

The compiler adds the following bridge method in `SubB1` in order for overriding to work properly at runtime:

Click here to view code image

```
public void set(Object t) {set((Number)t);}      // (1')
```

It does not add a bridge method for the `get()` method in `SubB1` because of covariant return: The return type `Number` for the method `get()` in `SubB1` is a subtype of the return type `Object` of the method `get()` in `SupB`.

Example 11.14 *Overriding from Generic Supertype*

Click here to view code image

```
class SupB<T> {
  public void set(T t) {/*...*/}                 // (1)
  public T get()       {return null;}            // (2)
}

class SubB1 extends SupB<Number> {
  @Override public void set(Number num) {/*...*/} // (1a) Overrides
  @Override public Number get()       {return 0;} // (2a) Overrides
}

class SubB2 extends SupB<Number> {
  @Override public void set(Object obj) {/*...*/} // (1b) Error: same erasure
  @Override public void set(Long l)     {/*...*/} // (1c) Error: overloads

  @Override public Object get() {          // (2b) Error: incompatible return type
    return null;
  }
}
```

We now examine the methods in `SubB2`. The `set()` method at (1b) has the *same signature* as *the erasure* of the signature of the `set()` method at (1) in the supertype `SupB`. If overriding were allowed, the bridge method added would result in *two* methods with the same signature `set(Object)` in `SubB2`. Two methods with the same signature are not permitted in the same class—called a *name clash*—therefore, (1b) is not allowed.

The method `set()` at (1c) is overloaded because its signature is different from the other `set()` methods in `SubB2` and `SupB`. It would compile if the `@Override` annotation was

removed. The method `get()` at (2b) has the return type `Object`, while the `get()` method in `SupB<Number>` has the return type `Number`. The return types are not covariant, and (2b) is rejected.

Example 11.15 shows a typical error where a generic supertype is extended, but its parameterization is missing in the `extends` clause of the subtype, as shown at (2). The `set()` method in `SubZ` neither overrides nor overloads the method at (1). Both methods have the same signature after erasure: s`et(Object)`. Adding a bridge method in `SubZ` to make the overriding work properly would result in a name clash. (1a) is rejected.

Example 11.15 *Missing Supertype Parameterization*

Click here to view code image

```
class SupZ<T> {
  public void set(T t) {/*...*/}            // (1)
}

class SubZ<E> extends SupZ {                // Superclass not parameterized
  @Override public void set(E e) {/*...*/}  // (1a) Error: same erasure
}
```

Genericity and Inherited Methods

The subsignature requirement for overriding means that the signature of the subtype method must be the *same* as that of the supertype method, or it must be the same as the *erasure* of the signature of the supertype method. Note the implication of the last sentence: The signature of the subtype method must be the same as the *erasure of the supertype method*, *not* the other way around. The converse is neither overloading nor overriding, but a *name clash* and is reported as an error.

The subsignature requirement also implies that a *generic subtype method cannot override a non-generic supertype method*. In other words, genericity cannot be added to an inherited method. This case is illustrated in **Example 11.16**. It is the erasures of the signatures of the generic methods in the subtype that are the same as the signatures of the non-generic methods in the supertype. Overriding requires the converse. A name clash is generally the reason why neither overriding nor overloading is permitted.

Example 11.16 *Genericity Cannot be Added to Inherited Methods*

Click here to view code image

```
class SupJ {
  public void set(Object obj) {/*...*/}          // (1)
  public Object get()    {return null;}          // (2)
```

```
}

class SubJ extends SupJ {
  @Override public <T> void set(T t) {/*...*/}   // (1a) Error: same erasure
  @Override public <S> S get()  {return null;}   // (2a) Error: same erasure
}
```

Overriding Abstract Methods from Multiple Superinterfaces

The code below illustrates the simple case where the subinterface `TaskAB` at (3) extends the two superinterfaces, `TaskA` and `TaskB`, whose abstract methods are *override-equivalent*—that is, they have the same signature. The abstract method declared at (4) in the subinterface `TaskAB` *overrides* the two abstract method declarations from the superinterfaces. This method represents the two abstract method declarations from the superinterfaces. The compiler can also infer it from the inherited methods, and its declaration at (4) can be omitted.

Click here to view code image

```
interface TaskA { Object compute(Integer iRef1); }   // (1)
interface TaskB { Object compute(Integer iRef2); }   // (2)
interface TaskAB extends TaskA, TaskB {              // (3)
  @Override Object compute(Integer iRef3);           // (4) Can be omitted.
}

interface TaskC { String compute(Integer iRef4); }   // (5)
interface TaskABC extends TaskA, TaskB, TaskC {      // (6)
  @Override String compute(Integer iRef5);           // (7a) Can be omitted.
//@Override Object compute(Integer iRef6);           // (7b) Compile-time error!
}
```

The subinterface `TaskABC` at (6) extends the superinterfaces `TaskA`, `TaskB`, and `TaskC`. The abstract methods of the three superinterfaces are override-equivalent, since they have the same method signature: `compute(Integer)`. The abstract method at (5) in `TaskC` can *override* the other two methods with its *covariant* return, as `String` type is a subtype of `Object` type. The compiler infers that the abstract method at (5) can represent the methods from the superinterfaces in the subinterface `TaskABC`. Again we can omit the abstract method declaration at (7a), as the compiler can infer it. A client of course must make sure that it implements the correct abstract method when implementing the interface `TaskABC`.

If the abstract method declaration at (7a) is replaced with the abstract method declaration at (7b) where the return type is `Object`, the code will not compile, as this method cannot override the abstract method declaration at (5) declared with the return type `String`. This is an example of a *name clash*—that is, the inherited methods from the superinter-

faces are override-equivalent, but the method at (7b) cannot override them all, thus resulting in a compile-time error.

The next example shows that the multiple abstract methods inherited by the `ContractZ` are not override-equivalent—in fact they are overloaded, and both are inherited from their respective interfaces by the subinterface.

Click here to view code image

```
interface ContractX { void doIt(int i); }
interface ContractY { void doIt(double d); }
interface ContractZ extends ContractX, ContractY {
  @Override void doIt(int d);                          // Can be omitted.
  @Override void doIt(double d);                       // Can be omitted.
}
```

So far all examples have involved methods without generics, where a subtype method could override a supertype method that had the same signature.

A subtype method *without generics* can *override* supertype methods *with generics* that have the same signature *after type erasure*. After type erasure of the `bake()` methods at (1) and (2) in the code below, all three `bake()` methods at (1), (2), and (3) are override-equivalent with the signature `bake(List)`. However, it is the `bake()` method at (3) that *overrides* the `bake()` methods at (1) and (2). This `bake()` method *without generics* at (3) logically represents all the inherited `bake()` methods in the `Bakeable` subinterface. The implementation of the `Bakeable` subinterface thus requires implementation of the `bake()` method without generics declared at (3).

Click here to view code image

```
class Ingredient { }

interface PizzaBakeable {
  void bake(List<Ingredient> ingredients);  // (1) After type erasure: bake(List)
}

interface CalzoneBakeable {
  void bake(List<Ingredient> ingredients);  // (2) After type erasure: bake(List)
}

interface SimplyBakeable {
  void bake(List ingredients);              // (3) Signature: bake(List)
}

interface Bakeable extends SimplyBakeable, PizzaBakeable, CalzoneBakeable {
```

```
    @Override void bake(List ingredients);    // Can be omitted.
}
```

In the code above, the `bake()` method without generics at (3) that *overrides* the other two inherited `bake()` methods, and thus represents all the inherited `bake()` methods in the subinterface, can be inferred by *type erasure of any of the inherited* `bake()` *methods with generics*, since they are all override-equivalent.

From the examples in this subsection, we see that override-equivalent methods in superinterfaces can be legally inherited by a subinterface, if a method can be inferred that *overrides* all inherited override-equivalent methods (after any type erasure), and which then represents these methods in the subinterface. In the case of override-equivalent methods *with generics* from multiple superinterfaces, the type erasure of *any* of the override-equivalent methods is always a candidate that can override and represent these methods in the subinterface.

11.13 Limitations and Restrictions on Generic Types

In this section we take a look at implications and restrictions on generic types for instance tests, casting, arrays, variable arity parameters, exception handling, nested classes, and enum types.

Reifiable Types

Concrete parameterized types are used by the compiler and then translated by erasure to their raw types, losing information about the parameterization in the process. In other words, only the raw types of these concrete parameterized types are available at runtime. For example, `List<Integer>` and `List<String>` are both erased to the raw type `List`. The same applies to unbounded parameterized types: `List<E>` is erased to `List`.

Non-generic types are not affected by type erasure, and therefore, have *not* lost any information and are, therefore, available fully at runtime. For example, the types `Integer` and `String` remain intact and are present unchanged at runtime.

Types that are completely available at runtime are known as *reifiable types*—that is, type erasure has *not* removed any important information about them (see **Table 11.5**). Types whose information has been affected by erasure are called *nonreifiable types* (see **Table 11.6**).

Note that unbounded wildcard parameterized types (`Node<?>` , `MyStack<?>`) are reifiable, whereas concrete parameterized types (`Node<Number>` , `MyStack<String>`) and bounded wildcard parameterized types (`Node<? extends Number>` , `MyStack<? super String>`) are non-reifiable.

As we shall see in the rest of this section, certain operations in Java are only permitted on reifiable types (as their type information is fully intact and available at runtime), and not on non-reifiable types (as their type information is not fully available at runtime, since it has been affected by type erasure).

Table 11.5 *Examples of Reifiable Types*

Reifiable type	Example
A primitive type	`int, double, boolean`
A non-generic type	`Exception, System, Math, Number`
A raw type	`List, ArrayList, Map, HashMap`
A parameterized type in which all type arguments are *unbounded wildcards* (unbounded wildcard parameterized type)	`List<?>, ArrayList<?>, Map<?,?>, HashMap<?,?>`
An *array type* whose component type is reifiable	`double[], Number[], List[], HashMap<?,?>[], Number[][]`

Table 11.6 *Examples of Non-Reifiable Types*

Non-Reifiable type	Example
A type parameter	`E, T, K, V`
A parameterized type with *concrete or unbounded type parameters* (concrete or unbounded parameterized type)	`List<E>, List<String>, ArrayList<Integer>, HashMap<String, Number> Map<K, V>`
A parameterized type with a *bound* (bounded wild-card parameterized type)	`List<? extends Object>, ArrayList<? extends Number>,`

	`Comparable<? super Integer>`
An *array type* whose component type is non-reifiable	`List<E>[]`, `ArrayList<Number>[]`, `Comparable<? super Integer>[]`, `HashMap<K, V>[]`

Implications for the `instanceof` operator

Although the discussion here is about the `instanceof` type comparison operator, it applies equally to the `instanceof` pattern match operator.

At (1) below, we want to determine whether the object referenced by `obj` is an instance of the concrete parameterized type `MyStack<Integer>`—that is, whether it is a stack of `Integer`.

Click here to view code image

```
Object obj;
...
boolean isIntStack = obj instanceof MyStack<Integer>; // (1) Compile-time error!
```

The post-erasure code for (1) is equivalent to the following statement:

Click here to view code image

```
boolean isIntStack = obj instanceof MyStack;          // (1')
```

The statement at (1') cannot perform the `instanceof` type comparison as expected at (1), since the type erasure has removed the information about the concrete type parameter `Integer`—that is, the type `MyStack<Integer>` is non-reifiable. The compiler issues an error because the type `Object` cannot be *safely cast* to the type `MyStack` at runtime.

In the following code, the fact that `IStack<Integer>` is a supertype of `MyStack<Integer>` can be checked at compile time—that is, the type `MyStack<Integer>` can be considered a *refinement* of the type `IStack<Integer>`. Such refinement of a generic type can be checked with the `instanceof` type comparison operator as it is deemed safe at runtime.

Click here to view code image

```
IStack<Integer> iStack;
...
```

```
boolean isIntegerStack = iStack instanceof MyStack<Integer>;   // (2) OK.
```

The post-erasure code for (2) is equivalent to the following statement, where `My-Stack` is a subtype of `IStack` (i.e., the type of `iStack`), making the cast from `IStack` to `MyStack` safe.

Click here to view code image

```
boolean isIntegerStack = iStack instanceof MyStack;   // (2')
```

Given that `T` is a formal type parameter, the following code will not compile, as the arguments of the `instanceof` type comparison operator are non-reifiable types.

Click here to view code image

```
boolean isT      = obj instanceof T;                  // (3) Compile-time error!
boolean isTStack = obj instanceof MyStack<T>;         // (4) Compile-time error!
```

The post-erasure code for (2) and (3) is equivalent to the following statements. Again, casting the type `Object` to the types `Object` and `MyStack`, respectively, is not deemed safe at runtime.

Click here to view code image

```
boolean isT      = obj instanceof Object;             // (3')
boolean isTStack = obj instanceof MyStack;            // (4')
```

If we just wanted to determine that an instance was some stack, the instance test can be performed against the raw type or the unbounded wildcard parameterized type, as these types are reifiable:

Click here to view code image

```
boolean isRawStack = obj instanceof MyStack;
boolean isAnyStack = obj instanceof MyStack<?>;       // Preferable.
```

Implications for Casting

A non-reifiable type can lose important type information during erasure and the cast may not have the desired effect at runtime. A cast to a non-reifiable type is *generally* flagged as an *unchecked cast warning*, and the cast is *replaced by a cast to its erasure*. Again, the compiler permits casts to allow interoperability between legacy code and generic code—usually with a warning.

The following code shows why a warning is necessary. The reference value of a `Number` node, declared at (1), is assigned to a reference of type `Node<?>` at (2). This reference is cast to a `Node<String>` and its reference value is assigned to a reference of type `Node<String>` at (3). A `String` is set as data in the node at (4). The data is retrieved from the node via the `numNode` reference and assigned to a `Number` reference at (5).

Click here to view code image

```
Node<Number> numNode = new Node<>(20, null);       // (1)
Node<?> anyNode = numNode;                         // (2)
Node<String> strNode = (Node<String>) anyNode;     // (3) Unchecked cast warning
strNode.setData("Peekaboo");                       // (4)
Number num = numNode.getData();                    // (5) ClassCastException
```

The erasure of the assignment at (3) is equivalent to the following assignment, with the cast succeeding at runtime:

Click here to view code image

```
Node strNode = (Node) anyNode;                     // (3')
```

However, a `ClassCastException` occurs at (5) because a `String` cannot be assigned to a `Number`. The compiler warns of potential problems by issuing an unchecked cast warning at (3).

The types `Node<String>` and `Node<Number>` are unrelated. That is the reason why the `Number` node in the above example was compromised by going through a node of type `Node<?>`. As we would expect, a cast between unrelated types results in a compile-time error:

Click here to view code image

```
strNode = (Node<String>) numNode;                  // Compile-time error
```

If we are casting a generic supertype to a generic subtype, where the parameterization is identical, the cast is safe and no warning is issued:

Click here to view code image

```
// BiNode<E> is a subtype of MonoNode<E>.
MonoNode<String> monoStrNode = new BiNode<>("Hi", null, null);
BiNode<String> biStrNode = (BiNode<String>) monoStrNode; // Ok. No warning.
```

The method `castaway()` below shows examples of casting an `Object` reference that refers to a node of type `String`, declared at (2).

Click here to view code image

```
//@SuppressWarnings("unchecked")         // (1) Suppress warnings at (4),(6),(7).
public static void castaway() {
  Object obj = new Node<>("one", null);              // (2)
  Node<String> node1 = obj;                          // (3) Compile-time error!
  Node<String> node2 = (Node<String>) obj;           // (4) Unchecked cast
  Node<String> node3 = (Node<?>) obj;                // (5) Compile-time error!
  Node<String> node4 = (Node<String>)(Node<?>) obj;  // (6) Unchecked cast
  Node<String> node5 = (Node) obj;                   // (7) Unchecked conversion
  Node<?> node6 = (Node) obj;                        // (8) OK.
  Node<?> node7 = (Node<?>)obj;                      // (9) OK.
}
```

It is instructive to see what warnings and errors are issued by the compiler. The compile-time error at (3) is due to incompatible types: An `Object` cannot be assigned to a `Node<String>` reference. The compiler issues an *unchecked cast warning* at (4) because of the cast from an `Object` to the concrete parameterized type `Node<String>`. The compile-time error at (5) is due to incompatible types: A `Node<?>` cannot be assigned to a `Node<String>` reference. There are two casts at (6): An `Object` is cast to `Node<?>`, which in turn is cast to `Node<String>`. The cast to `Node<?>` is permitted, but the second cast results in an unchecked cast warning. The compiler issues an *unchecked conversion warning* at (7), since a raw type (`Node`) is being assigned to a parameterized type (`Node<String>`). (8) and (9) show that casting to the raw type or to the unbounded wildcard is always permitted, since both types are reifiable.

If the annotation `@SuppressWarnings("unchecked")` at (1) is uncommented, the unchecked warnings at (4), (6), and (7) in the method `castaway()` will be suppressed (§25.5, p. 1582). Use of this annotation is recommended when we *know* that unchecked cast warnings are inevitable in a language construct (a type declaration, a field, a method, a parameter, a constructor, a local variable). Any unchecked warnings reported by the compiler are those that were *not* documented using this annotation. The use of an unbounded wildcard is recommended in casts, rather than using raw types, as it provides for stricter type checking.

Implications for Arrays

Array store checks are based on the element type being a reifiable type, in order to ensure that subtype covariance between array types is not violated at runtime. In the code below, the element type of the array is `String` and the array store check at (1) disallows the assignment, resulting in an `ArrayStoreException` because the reference value of a `Double` cannot be stored in a `String` reference.

Click here to view code image

```
String[] strArray = new String[] {"Hi", "Hello", "Howdy"};
Object[] objArray = strArray; // String[] is a subtype of Object[]
objArray[0] = 2020.5;         // (1) ArrayStoreException
```

We cannot instantiate a formal type parameter, nor can we create an array of such a type:

Click here to view code image

```
// T is a formal type parameter.
T t = new T();                // Compile-time error!
T[] anArray = new T[10];      // Compile-time error!
```

It is also not possible to create an array whose element type is a concrete or a bounded wildcard parameterized type:

Click here to view code image

```
// An array of Lists of String
List<String>[] list1 = {                          // Compile-time error
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

List<String>[] list2 = new List<String>[] {       // Compile-time error
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

// An array of Lists of any subtype of Number
List<? extends Number>[] list3
               = new List<? extends Number>[] {   // Compile-time error

  Arrays.asList(20.20, 60.60), Arrays.asList(1948, 1949)
};
```

Unbounded wildcard parameterized types are allowed as element types because these types are essentially equivalent to the raw types (p. 584):

Click here to view code image

```
List<?>[] list4 = {
  Arrays.asList("one", "two"), Arrays.asList("three", "four")
};

List<?>[] list5 = new List<?>[] {
  Arrays.asList(20.20, 60.60), Arrays.asList(1978, 1981)
```

```
};
List[] list6 = list5;
```

Note that we can always declare a *reference* of a non-reifiable type. It is creating arrays of these types that is not permitted.

Click here to view code image

```
class MyIntList extends ArrayList<Integer> {}     // A reifiable subclass.

// Client code
List<Integer>[] arrayOfLists = new MyIntList[5];  // Array of Lists of Integer
List<Integer[]> listOfArrays = new ArrayList<>(); // List of Arrays of Integer
```

The class `MyStack<E>` in **Example 11.10**, **p. 598**, implements a method to convert a stack to an array:

Click here to view code image

```
// Copy to array as many elements as possible.
public E[] toArray(E[] toArray) {                              // (11)
  Node<E> thisNode = tos;
  for (int i = 0; thisNode != null && i < toArray.length; i++) {
    toArray[i] = thisNode.getData();
    thisNode = thisNode.getNext();
  }
  return toArray;
}
```

Note that the array is passed as a parameter because we cannot create an array of the parameter type, as the following version of the method shows:

Click here to view code image

```
public E[] toArray2() {
  E[] toArray = new E[numOfElements];            // Compile-time error
  int i = 0;
  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

The third version below uses an array of `Object`. The cast is necessary in order to be compatible with the return type. However, the cast is to a non-reifiable type, resulting in an unchecked cast warning:

Click here to view code image

```
public E[] toArray3() {
  E[] toArray = (E[])new Object[numOfElements];  // (1) Unchecked cast warning
  int i = 0;

  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

The method implementation above has a serious problem, even though the code compiles. We get a `ClassCastException` at (2) below because we cannot assign the reference value of an `Object[]` to an `Integer[]` reference:

Click here to view code image

```
MyStack<Integer> intStack = new MyStack<>();
intStack.push(9); intStack.push(1); intStack.push(1);
Integer[] intArray = intStack.toArray3();        // (2) ClassCastException
```

The final and correct version of this method uses *reflection* to create an array of the right element type:

Click here to view code image

```
@SuppressWarnings("unchecked")
public E[] toArray4(E[] toArray) {
  if (toArray.length != numOfElements) {
    toArray = (E[])java.lang.reflect.Array.newInstance(             // (3)
                         toArray.getClass().getComponentType(),
                         numOfElements);          // Suppressed unchecked warning
  }
  int i = 0;
  for (E data : this) { toArray[i++] = data; }
  return toArray;
}
```

The method is passed an array whose element type is determined through reflection, and an array of this element type (and right size) is created at (3). The method `newInstance()` of the `Array` class creates an array of specified element type and size. The element type is looked up through the class literal of the array supplied as an argument. The unchecked cast warning is suppressed because we know it is unavoidable. We will not go into the nitty-gritty details of using reflection here.

The client code now works as expected. We pass an array of zero length, and let the method create the array.

Click here to view code image

```
MyStack<Integer> intStack = new MyStack<>();
intStack.push(9); intStack.push(1); intStack.push(1);
Integer[] intArray = intStack.toArray4(new Integer[0]);              // OK.
```

The next example demonstrates the danger of casting an array of a reifiable type to an array of a non-reifiable type. An array of the raw type `List` (reifiable type) is created at (1), and cast to an array of `List<Double>` (non-reifiable type). The cast results in an unchecked cast warning. The first element of the array of `List<Double>` is initialized with a list of `Double` at (2). The reference value of this array is assigned to a reference of type `List<? extends Number>` at (3). Using this reference, the list of `Double` in the first element of the array is replaced with a list of `Integer` at (4). Using the alias `arrayOfListsOfDouble` of type `List<Double>[]`, the first element in the first list of the array (an `Integer`) is assigned to a `Double` reference. Since the types are incompatible, a `ClassCastException` is thrown at (5). Note that the array store check at (4) succeeds because the check is against the reified element type of the array, `List`, and not `List<Double>`.

Click here to view code image

```
List<Double>[] arrayOfListsOfDouble
               = (List<Double>[]) new List[1];    // (1) Unchecked cast warning!
arrayOfListsOfDouble[0] = Arrays.asList(10.10);   // (2) Initialize
List<? extends Number>[] arrayOfListsOfExtNums = arrayOfListsOfDouble; // (3)
arrayOfListsOfExtNums[0] = Arrays.asList(10);     // (4) Array storage check ok
Double firstOne = arrayOfListsOfDouble[0].get(0); // (5) ClassCastException!
```

Implications for Non-Reifiable Variable Arity Parameter

Because variable arity parameters are treated as arrays, generics have implications for non-reifiable variable arity parameters (`T ... varargs`). Most of the workarounds for arrays are not applicable, as array creation is implicit for variable arity parameters. In a method declaration with a non-reifiable variable arity parameter, the compiler flags a warning about possible *heap pollution* from using a non-reifiable variable arity type. Heap pollution occurs when a reference of a parameterized type refers to an object that is not of the parameterized type. This can only occur when both the compiler and the JVM cannot guarantee the validity of an operation involving parameterized types. The compiler issues a warning about potential heap pollution, and the JVM normally throws an appropriate exception.

The method `asStack()` below has a variable arity parameter at (1) whose type is a non-reifiable type `T`. The method pushes the specified `elements` on to the specified `stack`. The compiler issues a possible heap pollution warning at (1).

Click here to view code image

```
public static <T> void
  asStack(MyStack<T> stack, T...elements) { // (1) Possible heap pollution warning
  for (T element : elements) { stack.push(element); }
}
```

In a method call, implicit creation of a *generic array with the variable arity parameter* results in an *unchecked generic array creation warning*—and type-safety is no longer guaranteed. The method above is called by the client code below at (4). The idea is to initialize a stack of stacks of `Integer` with a stack of `Integer`. An implicit generic array (`new MyStack[] { intStack }`) is created by the compiler, which is passed in the method call at (4). The compiler also issues an *unchecked array creation warning*, but the code compiles and runs without any problems.

Click here to view code image

```
/* Client code */
// (2) Create a stack of stacks of Integer:
MyStack<MyStack<Integer>> stackOfIntStacks = new MyStack<>();

// (3) Create a stack of Integer:
MyStack<Integer> intStack = new MyStack<>();
intStack.push(2019); intStack.push(2020);

// Initializes the stack of stacks with the stack of Integer.
MyStack.asStack(stackOfIntStacks, intStack); // (4) Unchecked array creation!
intStack = stackOfIntStacks.pop();      // (5) Pop the stack of stacks of Integer.
int tos = intStack.pop();               // (6) Pop the stack of Integer.
assert tos == 2020;
```

The implicit array passed as an argument is available as an array of a *non-reifiable type* in the body of the method `asStack()`. The integrity of this array can be compromised by making the array store check report a false positive at runtime—that is, succeed when the store operation should normally fail, thereby resulting in heap pollution. This is demonstrated by the method declaration below, in the assignment statement at (1a), where the contents of the `elements` array are changed before they are copied to the specified `stack`. The compiler issues a possible heap pollution warning at (1) and an unchecked cast warning at (1a).

Click here to view code image

```
public static <T> void asStackMalicious(
         MyStack<T> stack, T... elements) { // (1) Possible heap pollution warning
  // Compromise the elements array:
  MyStack<Double> doubleStack = new MyStack<>();
  doubleStack.push(20.20);
  elements[0] = (T) doubleStack;            // (1a) Unchecked cast warning!
```

```
  // Copy from elements array:
  for (T element : elements) { stack.push(element); }
}
```

A partial erasure for the method `asStackMalicious()` is shown below.

Click here to view code image

```
public static void asStackMalicious(MyStack stack, Object...elements) {
  // Compromise the elements array:
  MyStack doubleStack = new MyStack();
  doubleStack.push(Double.valueOf(20.20));
  elements[0] = (Object) doubleStack;                    // (1b)
  for (Object element : elements) { stack.push(element); } // (1c)
...
}
```

Note that the *cast* at (1b) succeeds for *any* object at runtime, as any object can be cast to `Object`. The assignment at (1b) succeeds if the array store check succeeds.

If we now call the method `asStackMalicious()`, instead of the method `asStack()` at (4) in the client code above, the code compiles with a generic array creation warning as before.

Click here to view code image

```
MyStack.asStackMalicious(stackOfIntStacks, intStack); // (4') Unchecked warning!
```

At runtime, the reference `elements` in the method `asStackMalicious()` refers to the implicit array created in the call—that is, `new MyStack[] { intStack }`. The signature of the method call at runtime is equivalent to:

Click here to view code image

```
asStackMalicious(MyStack, MyStack[])
```

At runtime, the actual type of the `stack` parameter is `MyStack` and the actual type of the `elements` array parameter is `MyStack[]`. The element type of a stack has been erased. The references `doubleStack` and `elements[0]` both have the runtime type `MyStack`. When the code is run, the array store check succeeds at (1b), and `element[0]` now refers to a stack of `Double`. In the loop at (1c), this stack of `Double` is pushed on the stack of stacks of `Integer` referred to by the formal parameter `stack`. Heap pollution is now a fact.

After return from the call at (4) in the client code, the assignment at (5) also succeeds, as this is an assignment of a `MyStack` to a reference of the same type after erasure. The reference `intStack` now refers to a stack of `Double`. The error is only discovered after a `ClassCastException` is thrown at (6) because the `Double` that is popped from the stack of `Integer` cannot be converted and assigned to an `Integer`.

The general rule is to avoid the variable arity parameter in methods where the parameter is of a non-reifiable type. No matter what, it is the responsibility of the method declaration to ensure that the non-reifiable variable arity parameter is handled in a type-safe manner, and then to use one of the two annotations below to suppress unchecked warnings:

Click here to view code image

```
@SuppressWarnings("unchecked")                          // (1)
public static <T> void asStack(MyStack<T> stack, T...elements) { ... }

@SafeVarargs                                            // (2)
public static <T> void asStack(MyStack<T> stack, T...elements) { ... }
```

The method `asStack()` handles the non-reifiable variable arity parameter `elements` in a type-safe way, so suppressing the unchecked warning is justified. The annotation at (1) suppresses *all* unchecked warnings in the method declaration, but it does *not* suppress the unchecked generic array creation warning at the call sites. The annotation at (2) suppresses all unchecked warnings in the method declaration, as well as the unchecked generic array creation warning at the call sites. However, the `@SafeVarargs` annotation (§25.5, p. 1582) is only applicable to a variable arity method or constructor, and the method cannot be overridden—that is, the method must be either `static`, `private`, or `final`.

Implications for Exception Handling

When an exception is thrown in a `try` block, it is matched against the parameter of each `catch` block that is associated with the `try` block. This test is similar to the instance test, requiring reifiable types. The following restrictions apply, and are illustrated in **Example 11.17**:

- A generic type cannot extend the `Throwable` class.
- A parameterized type cannot be specified in a `throws` clause.
- The type of the parameter of a `catch` block must be a reifiable type, and it must also be a subtype of `Throwable`.

Example 11.17 *Restrictions on Exception Handling*

Click here to view code image

```
// File: ExceptionErrors.java

// (1) A generic class cannot extend Exception:
class MyGenericException<T> extends Exception { }        // Compile-time error!

public class ExceptionErrors {

  // (2) Cannot specify parameterized types in throws clause:
  public static void main(String[] args)
                    throws MyGenericException<String> { // Compile-time error!
    try {
      throw new MyGenericException<String>();
    } // (3) Cannot use parameterized type in catch block:
      catch (MyGenericException<String> e) {             // Compile-time error!
      e.printStackTrace();
    }
  }
}
```

However, *type parameters* are allowed in the `throws` clause, as shown in **Example 11.18**. In the declaration of the `MyActionListener` interface, the method `doAction() can` throw an exception of type `E`. The interface is implemented by the class `FileAction`, that provides the actual type parameter (`FileNotFoundException`) and implements the `doAction()` method with this actual type parameter. All is aboveboard, as only reifiable types are used for exception handling in the class `FileAction`.

Example 11.18 *Type Parameter in* `throws` *Clause*

Click here to view code image

```
public interface MyActionListener<E extends Exception> {
  public void doAction() throws E;       // Type parameter in throws clause
}
_____________________________________________________________________________
import java.io.FileNotFoundException;

public class FileAction implements MyActionListener<FileNotFoundException> {

  @Override public void doAction() throws FileNotFoundException {
     throw new FileNotFoundException("Does not exist");
  }

  public static void main(String[] args) {
    FileAction fileAction = new FileAction();
    try {
      fileAction.doAction();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
```

```
    }
  }
}
```

Implications for Nested Classes

Nested classes and interfaces can be declared as generic types, as shown in **Example 11.19**. All nested generic classes, except anonymous classes, can specify formal type parameters in their declaration, as at (2) through (6). Anonymous classes do not have a name, and a class name is required to declare a generic class and specifyits formal type parameters. Thus an anonymous class can only be declared as a parameterized type, where the actual type parameters are either specified in the anonymous class expression, as at (7) through (9), or can be inferred from the context when the diamond operator is used, as at (10).

Example 11.19 *Generic Nested Types*

Click here to view code image

```
class GenericTLC<A> {              // (1) Top-level class

  static class SMC<B> {/*...*/}    // (2) Static member class

  interface SMI<C> {/*...*/}       // (3) Static member interface

  class NSMC<D> {/*...*/}          // (4) Non-static member (inner) class

  void nsm() {
    class NSLC<E> {/*...*/}        // (5) Local (inner) class in non-static context
  }

  static void sm() {
    class SLC<F> {/*...*/}         // (6) Local (inner) class in static context
  }

  // Anonymous classes as parameterized types:
  SMC<A> xsf = new SMC<A>() {      // (7) In non-static context.
    /*...*/                       //     A is type parameter in top-level class.
  };
  SMC<Integer> nsf = new SMC<Integer>() {           // (8) In non-static context
    /*...*/
  };
  static SMI<String> sf = new SMI<String>() {       // (9) In static context
    /*...*/
  };
  SMC<String> nsf1 = new SMC<>() {                  // (10) Using diamond operator
    /*...*/
```

```
    };
  }
```

The type parameter names of a generic nested class can hide type parameter names in the enclosing context (see (2) in **Example 11.20**). Only a non-static nested class can use the type parameters in its enclosing context, as type parameters cannot be referenced in a static context.

Example 11.20 also illustrates instantiating generic nested classes. As a static member class does not have an outer instance, only its simple name is parameterized, and not the enclosing types, as shown by the code at (6). As a non-static member class requires an outer instance, any generic enclosing types must also be parameterized and instantiated, as shown by the code at (7). See §9.3, p. 502, for the syntax used in instantiating nested classes.

Example 11.20 *Instantiating Generic Nested Classes*

Click here to view code image

```
public class ListPool<T> {       // (1) Top-level class

  static class MyLinkedList<T> { // (2) Hiding type parameter in enclosing context
    T t;                         // T refers to (2)
  }

  class Node<E> {                // (4) Non-static member (inner) class
    T t;                         // T refers to (1)
    E e;
  }

  public static void main(String[] args) {
    // (5) Instantiating a generic top-level class:
    ListPool<String> lp = new ListPool<>();

    // (6) Instantiating a generic static member class:
    ListPool.MyLinkedList<String> list = new ListPool.MyLinkedList<>();

    // (7) Instantiating a generic non-static member class:
    ListPool<String>.Node<Integer> node1 = lp.new Node<>();
    ListPool<String>.Node<Double> node2 = lp.new Node<>();
    ListPool<Integer>.Node<String> node3
                      = new ListPool<Integer>().new Node<>();
  }
}
```

Other Implications

Enum Types

Because of the way enum types are implemented using the `java.lang.Enum` class, we cannot declare a generic enum type:

Click here to view code image

```
enum CoinToss<C> { HEAD, TAIL; }                    // Compile-time error!
```

An enum type can implement a parameterized interface, just like a non-generic class can. **Example 11.21** shows the enum type `TripleJump` that implements the `Comparator<TripleJump>` interface.

Example 11.21 *Enum Implements Parameterized Generic Interface*

Click here to view code image

```
enum TripleJump implements java.util.Comparator<TripleJump> {
  HOP, STEP, JUMP;

  @Override public int compare(TripleJump a1, TripleJump a2) {
    return a1.compareTo(a2);
  }
}
```

Class Literals

Objects of the class `Class<T>` represent classes and interfaces at runtime. For example, an instance of the `Class<String>` represents the type `String` at runtime.

A class literal expression (using `.class` notation) can only use reifiable types as type parameters, as there is only one class object created for each reifiable type.

Click here to view code image

```
Class<Node> class0 = Node<Integer>.class;// Non-reifiable type. Compile-time error!
Class<Node> class1 = Node.class;        // Reifiable type. Ok.
```

The `getClass()` method of the `Object` class also returns a `Class` object. The actual result type of this object is `Class<? extends |T|>` where `|T|` is the erasure of the static type of the expression on which the `getClass()` method is called. The following code shows that all invocations of the generic type `Node<T>` are represented by a single class literal:

Click here to view code image

```
Node<Integer> intNode = new Node<>(2019, null);
Node<String> strNode = new Node<>("Hi", null);
Class<?> class2 = strNode.getClass();
Class<?> class3 = intNode.getClass();
assert class1 == class2;
assert class2 == class3;
```

Review Questions

11.11 What will be the result of compiling and running the following program?

Click here to view code image

```
import java.util.*;

public class RQ100_00 {
  public static void main(String[] args) {
    List<String> lst1 = new ArrayList<>();
    List<Integer> lst2 = new ArrayList<>();
    List<List<Integer>> lst3 = new ArrayList<>();
    System.out.print(lst1.getClass() + ", ");
    System.out.print(lst2.getClass() + ", ");
    System.out.println(lst3.getClass());
  }
}
```

Select the one correct answer.

a.

Click here to view code image

```
class java.util.ArrayList<String>, class java.util.ArrayList<Integer>, class
java.util.ArrayList<List<Integer>>
```

b.

Click here to view code image

```
class java.util.ArrayList, class java.util.ArrayList, class
java.util.ArrayList
```

c.

Click here to view code image

```
class java.util.List, class java.util.List, class java.util.List
```

d.

Click here to view code image

```
class java.util.List<String>, class java.util.List<Integer>, class
java.util.List<List<Integer>>
```

e. The program will fail to compile.

f. The program will compile, but it will throw an exception at runtime.

11.12 Which method header can be inserted at (1) so that the program will compile and runs without errors?

Click here to view code image

```
import java.util.*;
public class RQ100_02 {
  public static void main(String[] args) {
    List<String> lst = Arrays.asList("Java", "only", "promotes", "fun");
    Collection<String> resultList = delete4LetterWords(lst);
  }

  // (1) INSERT METHOD HEADER HERE
  {
    Collection<E> permittedWords = new ArrayList<>();
    for (E word : words) {
      if (word.length() != 4) permittedWords.add(word);
    }
    return permittedWords;
  }
}
```

Select the one correct answer.

a.

Click here to view code image

```
static <E extends CharSequence>
Collection<? extends CharSequence> delete4LetterWords(Collection<E> words)
```

b.

Click here to view code image

```
static <E extends CharSequence>
List<E> delete4LetterWords(Collection<E> words)
```

c.

Click here to view code image

```
static <E extends CharSequence>
Collection<E> delete4LetterWords(Collection<? extends CharSequence> words)
```

d.

Click here to view code image

```
static <E extends CharSequence>
List<E> delete4LetterWords(Collection<? extends CharSequence> words)
```

e.

Click here to view code image

```
static <E extends CharSequence>
Collection<E> delete4LetterWords(Collection<E> words)
```

f.

Click here to view code image

```
static <E super CharSequence>
Collection<E> delete4LetterWords(Collection<E> words)
```

11.13 Which method declarations cannot be inserted independently at (2) to overload the method at (1)?

Click here to view code image

```
import java.util.*;

public class RQ_Overloading {

  static <T> void overloadMe(List<T> s1, List<T> s2) {} // (1)
  // (2) INSERT METHOD DECLARATIONS HERE
}
```

Select the two correct answers.

a.

Click here to view code image

```
static <T> void overloadMe(Collection<T> s1, List<T> s2) {}
```

b.

Click here to view code image

```
static <T> void overloadMe(List<T> s1, List<? extends T> s2) {}
```

c.

Click here to view code image

```
static <T> void overloadMe(List<T> s1, Collection<? super T> s2) {}
```

d.

Click here to view code image

```
static <T> void overloadMe(Collection<T> s1, Collection<? super T> s2) {}
```

e.

Click here to view code image

```
static <T> void overloadMe(Collection<T> s1, List<? super T> s2) {}
```

f.

Click here to view code image

```
static <T> void overloadMe(List<? extends T> s1, List<? super T> s2) {}
```

11.14 Which method declaration can be inserted at (1) so that the program will compile without warnings?

Click here to view code image

```
import java.util.*;
public class RQ100_87 {
  public static void main(String[] args) {
    List raw = new ArrayList();
    raw.add("2020");
    raw.add(2021);
    raw.add("2022");
    justDoIt(raw);
  }
  // (1) INSERT METHOD DECLARATION HERE
}
```

Select the one correct answer.

a. `static void justDoIt(List<Integer> lst) {}`

b. `static void justDoIt(List<?> lst) {}`

c. `static <T> void justDoIt(List<T> lst) {}`

d. None of the above

11.15 Which of the following statements are true about the following code?

Click here to view code image

```
class SupC<T> {
  public void set(T t) {/*...*/}       // (1)
  public T get() {return null;}        // (2)
}

class SubC1<M,N> extends SupC<M> {
  public void set(N n) {/*...*/}       // (3)
  public N get() {return null;}        // (4)
}

class SubC2<M,N extends M> extends SupC<M> {
  public void set(N n) {/*...*/}       // (5)
  public N get() {return null;}        // (6)
}
```

Select the four correct answers.

a. The method at (3) overloads the method at (1).

b. The method at (3) overrides the method at (1).

c. The method at (3) will fail to compile.

d. The method at (4) overloads the method at (2).

e. The method at (4) overrides the method at (2).

f. The method at (4) will fail to compile.

g. The method at (5) overloads the method at (1).

h. The method at (5) overrides the method at (1).

i. The method at (5) will fail to compile.

j. The method at (6) overloads the method at (2).

k. The method at (6) overrides the method at (2).

l. The method at (6) will fail to compile.

11.16 Which types cannot be declared as generic types? Select the three correct answers.

a. Enum types

b. Static member classes

c. Any subclass of `Throwable`

d. Nested interfaces

e. Anonymous classes

f. Non-static member classes

g. Local classes

11.17 Which statement is true about the following code?

Click here to view code image

```
import java.util.*;

public class CastAway {
  public static void main(String[] args) {
    Object obj = new ArrayList<Integer>();         // (1)
    List<?>       list1 = (List<?>) obj;           // (2)
    List<?>       list2 = (List) obj;              // (3)
    List          list3 = (List<?>) obj;           // (4)
    List<Integer> list4 = (List) obj;              // (5)
    List<Integer> list5 = (List<Integer>) obj;     // (6)
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile without any unchecked warnings. It will run with no output and terminate normally.

c. The program will compile without any unchecked warnings. When run, it will throw an exception.

d. The program will compile, but with unchecked warnings. It will run with no output and terminate normally.

e. The program will compile, but with unchecked warnings. When run, it will throw an exception.

11.18 Which statement is true about the following code?

Click here to view code image

```
import java.util.*;
public class InstanceTest2 {
  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    Set<Double> doubleSet = new HashSet<>();
    List<?>          list = intList;
    Set<?>            set = doubleSet;

    scuddle(intList);
    scuddle(doubleSet);
    scuddle(list);
    scuddle(set);
  }
  private static void scuddle(Collection<?> col) {
```

```
      if (col instanceof List<?>) {
        System.out.println("I am a list.");
      } else if (col instanceof Set<?>) {
        System.out.println("I am a set.");
      }
    }
  }
```

Select the one correct answer.

a. The method `scuddle()` will fail to compile.

b. The method `main()` will fail to compile.

c. The program will compile, but with an unchecked warning in method `scuddle()`. It will run and terminate normally with the following output:

```
I am a list.
```

```
I am a set.
```

```
I am a list.
```

```
I am a set.
```

d. The program will compile, but with an unchecked warning in the method `main()`. When run, it will throw an exception.

e. The program will compile without any unchecked warnings. It will run and terminate normally, with the following output:

```
I am a list.
```

```
I am a set.
```

```
I am a list.
```

```
I am a set.
```

f. The program will compile without any unchecked warnings. It will run and terminate normally, with the following output:

```
I am a list.
```

```
I am a set.
```

g. None of the above

11.19 Which statement is true about the following code?

Click here to view code image

```
public class GenArrays {
  public static <E> E[] copy(E[] srcArray) {
    E[] destArray = (E[]) new Object[srcArray.length];
    int i = 0;
    for (E element : srcArray) {
      destArray[i++] = element;
    }
    return destArray;
  }

  public static void main(String[] args) {
    String[] sa = {"9", "1", "1" };
    String[] da = GenArrays.copy(sa);
    System.out.println(da[0]);
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile, but with an unchecked warning. When run, it will print `9`.

c. The program will compile, but with an unchecked warning. When run, it will throw an exception.

d. The program will compile without any unchecked warnings. When run, it will print `9`.

e. The program will compile without any unchecked warnings. When run, it will throw an exception.

11.20 Which statement is true about the following code?

Click here to view code image

```
import java.util.*;
public class GenVarArgs {
  public static <T> void doIt(List<T>... aols) {          // (1)
    for (int i = 0; i < aols.length; i++) {
      System.out.print(aols[i] + " ");
    }
  }

  public static void main(String... args) {               // (2)
```

```
    List<String> ls1 = Arrays.asList("one", "two");
    List<String> ls2 = Arrays.asList("three", "four");
    List<String>[] aols = new List[] {ls1, ls2};          // (3)
    doIt(aols);                                           // (4)
  }
}
```

Select the one correct answer.

a. The program will fail to compile because of errors at (1).

b. The program will fail to compile because of errors at (2).

c. The program will fail to compile because of errors at (3).

d. The program will fail to compile because of errors at (4).

e. The program will compile with unchecked warnings. When run, it will print `[one, two] [three, four]`.