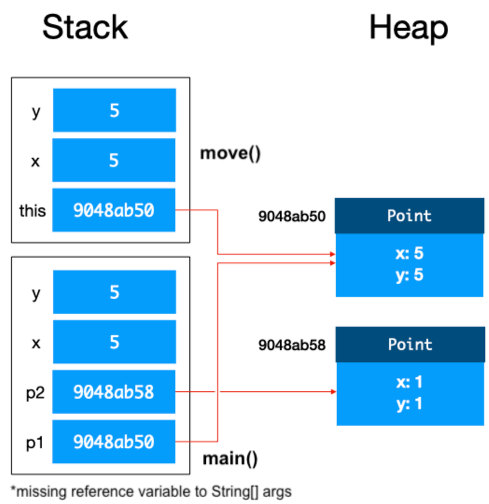
1 Stack and Heap

Consider the code block below:

```
class Point {
  private double x;
  private double y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public void move(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public static void main(String[] args) {
    Point p1 = new Point(0, 0);
    Point p2 = new Point(1, 1);
    double x = 5;
    double y = 5;
    p1.move(x, y);
  }
}
```



2 Information Hiding

```
class Circle {
  double x;
  double y;
  double r;

  double getArea() {
    return 3.141592653589793 * r * r;
  }
}
c.r = 10; // set the radius to 10
```

Consider the code above, where the radius of a `Circle c` is modified directly with `c.r = 10`. In doing so, the client to `Circle` makes an explicit assumption about the implementation of `Circle`. The implementation details have been leaked outside the abstraction barrier.

If the implementer wishes to change the representation of the `Circle` to store the diameter instead, that would invalidate the client's code. The client will have to carefully change all the code that makes this assumption, increasing the chances of introducing a bug.

2.1 Access Modifiers

A field or method declared as `private` cannot be accessed from outside the class, not even a child class. A `public` field or method can be accessed, modified, or invoked from outside the class.

Such a mechanism to protect the abstraction barrier from being broken is called data hiding or information hiding. This protection is enforced by the compiler at compile time.

Access Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`default`	Y	Y	N	N
`private`	Y	N	N	N

```java
class A {
    public void publicFunction() { ... }
    private void privateFunction() { ... }
}
class B extends A {
    // does not compile, method does not override any of its supertype's method
    @Override
    private void privateFunction() { ... }
}

B b = new B();
b.publicFunction(); // ok
b.privateFunction(); // not ok, symbol not found error
```

2.2 `final` Keyword

- A variable declared as `final` cannot be modified
  Note: The internal state of an object pointed to by a reference variable can be changed, only the variable cannot be re-bound to another object
- A method declared as `final` cannot be overriden
- A class declared as `final` cannot be inherited

3 Tell, Don't Ask

A class should provide methods to retrieve or modify the properties of the object. These methods are called the **accessor (getter)** and **mutator (setter)**. However, if there were getters and setters for every private field, then the internal representation would be exposed, thereby violating the encapsulation principle.

The right approach is to implement a method within the class that does whatever we want the class to do. For example, if we want to check if a given point `(x,y)` lies within a circle, one approach would be:

```
double cX = c.getX();
double cY = c.getY();
double r = c.getR();
boolean isInCircle = ((x - cX) * (x - cX) + (y - cY) * (y - cY)) <= r * r;
```

A better approach would be to add a new `boolean` method in the `Circle` class, and call it instead:

```
boolean isInCircle = c.contains(x, y);
```

This approach involves writing a few more lines of code to implement the method, but it keeps the encapsulation intact. If the implementer of `Circle` decided to change the representation of the circle and remove the direct accessors to the fields, then only the implementer needs to change the implementation of `contains()`. The client does not have to change anything.

The client should **tell** a `Circle` object what to do (e.g. compute the circumference), instead of **asking** for a value (e.g. radius) of a field and performing the computation on the object's behalf.

4 Subtypes

`A <: B` is read as "`A` is a subtype of `B`"
Tip: Subtype relationships can be made more intuitive by thinking in terms of:

- "`A` **extends** `B`"
- "`A` **is-a** `B`"
- "`A` **is more specific** than `B`"

The subtype relation is **reflexive and transitive**, i.e.

- `A <: A`
- `A <: B` and `B <: C` $\rightarrow$ `A <: C`

4.1 Variance

Variance refers to how subtyping between **complex types** relates to subtyping between their **components**.

Let `C(S)` corresponds to some complex type based on type `S`.

- `C` is covariant if `S <: T` $\rightarrow$ `C(S) <: C(T)`
- `C` is contravariant if `S <: T` $\rightarrow$ `C(S) :> C(T)`
- `C` is invariant if it is neither covariant nor contravariant.

4.1.1 Covariance of `T[]`

The Java Array is covariant. This means that `S <: T` $\rightarrow$ `S[] <: T[]`

```
Integer[] intArray;
Object[] objArray;
objArray = intArray; // ok
```

However, the implications of making an array covariant is the possibility of run-time errors:

```
Integer[] intArray = new Integer[2] {
  new Integer(10), new Integer(20)
};
Object[] objArray;
objArray = intArray;
objArray[0] = "Hello!"; // <- compiles!
```

On Line 5, `objArray` (with compile-time type of `Object[]`) is set to refer to an object with a run-time type of `Integer[]`. This is allowed since the array is covariant.

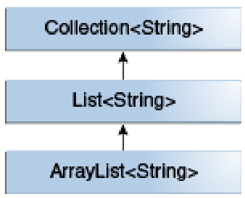
On Line 6, we try to put a `String` object into the `Object array`. Since `String <: Object`, the compiler allows this. The compiler does not realize that at run-time, the `Object` array will refer to an array of `Integer`.

Thus, the code compiles, but will crash when executing Line 6. This is an example of a type system rule that is unsafe.

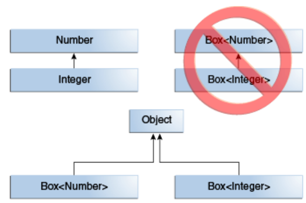
4.2 Variance of Generics and Wildcard

Covariance within Declared Classes
If `T <: S` then `T<X> <: S<X>`

Invariance within Generics
`X<T> and X<S>` are invariant. For example, a method accepting `ArrayList<Object>` will not accept an `ArrayList<Integer>`, even though `Integer <: Object`. This is because the compiler has no way of knowing at runtime the type information of the type parameters, due to type erasure. The use of wildcards overcomes this issue.



Covariance within declared classes



Invariance within generics

Covariance in Upper-bounded Wildcards
If `T <: S` then `X<T> <: X<? extends S>`

i.e. `T` must be more specific than `S`

```
ArrayList<? extends Number> x = new ArrayList<Integer>(); // Integer is more specific than Number
```

Contravariance in Lower-bounded Wildcards
If `T :> S` then `X<T> <: X<? super S>`

i.e. `T` must be less specific than `S`

```
ArrayList<? super Number> y = new ArrayList<Object>(); // Object is less specific than Number
```

Note: Of course, by reflexive property of subtypes, `X<S> <: X<? extends S>` and `X<S> <: X<? super S>`

Examples

```
// Use the transitive property to derive subtype relationships

// ok. ArrayList<String> <: List<String> AND List<String> <: List<?>
List<?> list = new ArrayList<String>();

// not ok, cannot instantiate an interface!
List<? super Integer> list = new List<Object>();

// ok, ArrayList<Object> <: List<Object> AND List<Object> <: List<? extends Object>
List<? extends Object> list = new ArrayList<Object>();

// ok, the compiler treats `list` as ArrayList<Integer>
List<? super Integer> list = new ArrayList<>()
```

5 Liskov Substitution Principle (LSP)

Let ϕ be a property provable about object x of type T. Then $\phi(y)$ should be true for object y of type S where $S <: T$. If $S <: T$, then object of type T can be replaced by an object of type S without changing the desirable property of the program.

- LSP helps us determine when overriding / inheritance is appropriate
- A subclass should not break the expectations set by the superclass. If a class `B` is substitutable for a parent class `A` then it should be able to pass all test cases of the parent class `A`.

Consider the following code:

```java
public class Restaurant {
  public static final int OPENING_HOUR = 1200;
  public static final int CLOSING_HOUR = 2200;

  public boolean canMakeReservation(int time) {
    return time <= CLOSING_HOUR && time >= OPENING_HOUR;
  }
}

public class LunchRestaurant extends Restaurant {
  private final int peakHourStart = 1200;
  private final int peakHourEnd = 1400;

  @Override
  public boolean canMakeReservation(int time) {
    if (time <= peakHourEnd && time >= peakHourStart) return false;
    else if (time <= CLOSING_HOUR && time >= OPENING_HOUR) return true;
    else return false;
  }
}

public class FastFoodRestaurant extends Restaurant {
  @Override
  public boolean canMakeReservation(int time) { return true; }
}

Restaurant r = new Restaurant();
r.canMakeReservation(1200) == true; // Is true, therefore test passes
r.canMakeReservation(2200) == true; // Is true, therefore test passes

Restaurant r = new LunchRestaurant();
r.canMakeReservation(1200) == true; // Is false, therefore test fails
r.canMakeReservation(2200) == true; // Is true, therefore test passes

Restaurant r = new FastFoodRestaurant();
r.canMakeReservation(1200) == true; // Is true, therefore test passes
r.canMakeReservation(2200) == true; // Is true, therefore test passes
```

In this example, the desirable property, or the expectation set, by the parent class `Restaurant` is that it is available for reservation between 12pm to 10pm. This expectation is broken by the subclass `LunchRestaurant`, thereby violating LSP.

However, the other subclass `FastFoodRestaurant`, does not violate LSP. All test cases that pass for `Restaurant` would pass for `FastFoodRestaurant`. Therefore, anywhere we use an object of type `Restaurant`, we can use `FastFoodRestaurant` without breaking any previously written code.

5.1 Does `<some code>` Violate LSP?

Yes, `<some code>` violates LSP. The `<subclass>` **changes the behavior of the** `<superclass>` , such that `<some desirable property>` **no longer holds for** `<subclass>`. Therefore, places in the program where `<superclass>` is used cannot simply be replaced by `<subclass>`.

6 Interface Segregation

- Clients should not be forced to depend on methods it does not use
- Interfaces should be minimal
- They should not force classes to implement methods they cannot
- They should not force clients to know of methods they do not require

Bad Example

```
class Shape {
    public void print() { .. }
    public void draw() { .. }
}
class Drawer {
    private Shape[] shapes;
    public void drawAll() {
        for(Shape shape : shapes) {
            shape.print();
            shape.draw();
        }
    }
}
```

Fixed Example

```
class Shape implements Drawable, Printable {
    @Override
    public void print() { .. }

    @Override
    public void draw() { .. }
}
class Drawer {
    private Drawable[] drawables;
    public void drawAll() {
        for (Drawable drawable : drawables) {
            drawables.draw();
        }
    }
}
```

7 Method Signature

A function/method signature consists of the following:

- Name
- Parameter Type(s)
- Number of Parameter(s)
- Order of Parameter(s)

```java
public static double f(Double x, Double y) { return x + y; }
public static double f(double x, Double y) { return x - y; }
public static double f(Double x, double y) { return x * y; }
/**
* the code above compiles without error.
* Each method has a different signature as Double and
* double are considered different types
*/

f(new Double(7.0), new Double(3.0)) // 10
f(7.0, new Double(3.0)) // 4
f(new Double(7.0), 3.0) // 21
f(7.0, 3.0) // run-time error due to ambiguity
```

7.1 Why Return Type can be Ignored

Notice that the signature **does not** require the method's **return type**. This is because Java supports covariant return types for overridden methods. This means an overridden method can have a more specific return type.

As long as <overriding return type> <: <overridden return type>, it's allowed.

Exam answer: "Existing code that has been written to invoke the superclass' method would still work if the code invokes the subclass' method instead after the subclass inherits from the superclass".

7.2 Method Overloading

Method overloading occurs when two functions have the **same name** but **different parameter types and/or number of parameters**.

7.3 Dealing with Multiple Overloaded Methods

The compiler tries to search for a method that exactly matches the input. If no such method is found, e.g. `f(1)` with the only method accepting a `double`, then type promotion (widening conversion) occurs.

If more than one function have equal suitability, an error is thrown.

```
|  Error:
|  reference to f is ambiguous
|  both methods f(int, double) and f(double, int) match a.f(1, 1)
```

7.4 Method Overriding

Method overriding occurs when a subclass implements a method with the **same signature** as a method in its superclass. It is recommended that overriding methods are **annotated** with `@Override`.

```
class A {
    public void f(int i, int j) { ... }
}
class B extends A {
    // this method does not override A's method!
    public void f(Integer i, Integer j) { ... }
}
class C extends A {
    // this method overrides A's method
    public void f(int i, int j) { ... }
}

B b = new B();
b.f(1, 1) // calls A's original method
b.f(new Integer(1), new Integer(2)) // calls B's method
```

While annotations do not affect the code, they are hints to the compiler that helps us detect errors early. `@Override` tells the compiler that the following method is intended to override a method in the parent class. In case there is a typo or overriding is not possible, the compiler will be able to detect it. When the compiler is not able to resolve the call/binding at compile time, it uses **dynamic** or **late binding**. Method overriding is a perfect example of dynamic binding as in this case, the **run-time type** of the object determines the method that is executed.

8 Polymorphism

Consider the code:

```
void say(Object obj) {
    System.out.println("Hi, I am " + obj.toString());
}
Point p = new Point(0, 0);
say(p); // "Hi, I am (0.0, 0.0)"
Circle c = new Circle(p, 4);
say(c); // "Hi, I am { center: (0.0, 0.0), radius: 4.0 }"
```

The same method invocation `obj.toString()` causes different methods to be called. The overriding `Point::toString` is invoked in the first call, and `Circle::toString` in the second call.

This happens even though the method receives an `Object` instance, since `Point <: Object` and `Circle <: Object`. The method that is invoked is decided during run-time, depending on the run-time type of the `obj`. This is known as dynamic or late binding.

Therefore, by **overriding** methods, we can take advantage of polymorphism. We can write general methods such as the `contains` method below, which will use the corresponding implementation of the `equals` method depending on the **run-time type** of `curr`.

```
boolean contains(Object array[], Object obj) {
  for (Object curr : array)
    if (curr.equals(obj)) return true;
  return false;
}
```

8.1 Dealing with Multiple Overriden Methods

Java's decision process to resolve which method implementation should be executed when a method is invoked, is a two-step process. The first occurs during compilation; the second during run time.

8.1.1 Compile Time

Let the compile-time type of the target be `C`. To determine the method descriptor, the compiler searches for all methods within `C` that can be correctly invoked on the given argument.

In the example above, it looks at the class `Object`, and there is only one method called `equals`. The method can be correctly invoked with one argument of type `Object`.

If there are multiple methods that can be correctly invoked, the compiler chooses the **most specific one**. Intuitively, a method `M` is more specific than method `N` if the arguments to `M` can be passed to `N` without compilation error. For example, if the class `Circle` implements:

```
boolean equals(Circle c) { .. }

@Override
boolean equals(Object c) { .. }
```

Then, `equals(Circle)` is more specific than `equals(Object)`. Every `Circle` is an `Object`, but not every `Object` is a `Circle`. `Circle <: Object`. Once the method is determined, the method's descriptor (**method signature + return type**) is stored in the generated bytecode.

8.1.2 Run-Time

During execution, when a method is invoked, the method descriptor from compile time is retrieved. Then, the run-time type of the target is determined. Let the run-time type of the target be `R`. Java then looks for an accessible method with the matching descriptor in `R`.

If no such method is found, the search will continue up the class hierarchy, first to the parent class of `R`, then to the grand-parent class of `R`, and so on, until the root of `Object`. The first method implementation with a matching method descriptor found will be the one executed.

Worked Example

```java
class A {
  void foo(A a) { System.out.println("class: A, parameter: A"); } // f1
 }
class B extends A {
  @Override
  void foo(A a) { System.out.println("class: B, parameter: A"); } // f2
  void foo(B a) { System.out.println("class: B, parameter: B"); } // f3
 }
class C extends B {
  void foo(C a) { System.out.println("class: C, parameter: C"); } // f4
}
class D extends C {
  @Override
  void foo(B a) { System.out.println("class: D, parameter: B"); } // f5
}
A a = new D(), B b = new D(), C c = new D(), D d = new D();
/**
* 1. Compile time type of a is A -> Look for functions within A matching the signature
* 2. Found only 1 matching function (f1) -> f1 stored
* 3. Run-time type of a is D -> Starting from D to A
*    looking for functions that match **f1** only!
* 4. Call first matching function -> f2
*/
a.foo(d); // class: B, parameter: A (foo(A a) is overridden)

/**
* 1. Compile time type of c is C -> Look for functions within C matching the signature
* 2. Found 3 matching functions (f2, f3, f4)
* 3. Store the most specific one -> f4 stored
* 3. Run-time type of c is D -> Starting from D to C
*    looking for functions that match **f4** only!
* 4. Call first matching function -> f4
*/
c.foo(d); // class: C, parameter: C (foo(C a) is most specific)
```

Try it yourself!

```java
/**
* 1. Let C and args[] be the compile time types of the target
*    and the arguments respectively
* 2. What method(s) named foo can C access? Which of these can be
*    invoked with args[]?
* 3. Which one is most specific? Call it foo*
* 4. Let R be the run-time type of the target
* 5. From R up to Object, what is the first method that matches the descriptor of foo*?
*/
b.foo(d); // class: D, parameter: B (foo(B a) is overridden)
d.foo(d); // class: C, parameter: C (D extends C)
c.foo(b); // class: D, parameter: B
```

8.1.3 Invocation of Class Methods

The description above applies to instance methods. Class methods, on the other hand, do not support dynamic binding. The method to invoke is resolved **statically during compile time**.

The same process in Step 1 is taken, but the corresponding method implementation in class `C` will always be executed during run-time, without considering the run-time type of the target.

9 Inheritance, Composition

Inheritance describe **is-a** relationships, while composition describe **has-a** relationships.

The keywords `extends` and `implements` are used to specify inheritance:

```
class A {}
class B extends A {} // ok
class C extends A, B {} // not ok, a class can only extend a single parent class
class D extends A implements I, J {} // ok
class E implements I, J extends A {} // not ok, 'extends' must precede 'implements'

interface I {}
interface J implements I {} // not ok, an interface does not use the 'implements' keyword
interface K extends I, J {} // ok
interface L extends A {} // not ok, an interface cannot extend a class
```

Note: AFAIK, the above rules also apply for abstract classes (can swap class with abstract class)

9.1 Interfaces

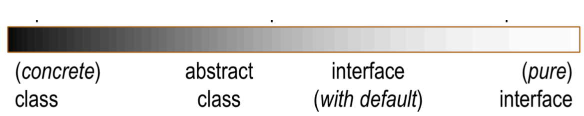
An interface is a contract between the two sides of the abstraction barrier. It is a template for a class that specifies how to create the class without any actual implementation.

9.2 Abstract Classes

A class which is declared as `abstract`. It can have abstract and non-abstract methods. It needs to be extended and its abstract methods implemented. It cannot be instantiated.

It can also have its own constructors and static methods, as well as `final` methods that stop subclasses from overriding them.

As long as there is a single abstract method in a class, the class needs to be declared as abstract.




```
class A {
    A() {} // ok
}
abstract class B {
    B() {} // ok
}
interface C {
    C() {} // not ok, interface cannot have any constructors
}

new A(); // ok
new B(); // not ok, an abstract class cannot be instantiated
new C(); // not ok, an interface cannot be instantiated
```

9.3 Default Methods for Interfaces

Prior to its introduction, if any interface needed additional methods, all implementing classes would need an implementation of their own of that method, even if it's a standardised one. As such, backward compatibility was not preserved.

```java
public interface Vehicle {
    String getBrand();
    default String turnAlarmOn() {
        return "Turning the vehicle alarm on";
    }
    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
 public class Car implements Vehicle {
    private String brand;
    // constructors/getters...
    @Override
    public String getBrand() { return brand; }
    public static void main(String[] args) {
        Vehicle car = new Car("BMW");
        System.out.println(car.getBrand()); // "BMW"
        System.out.println(car.turnAlarmOn()); // "Turning the vehicle alarm on."
        System.out.println(car.turnAlarmOff()); // "Turning the vehicle alarm off."
    }
}
```

9.3.1 Ambiguity in Default Methods

Since a single class can implement multiple interfaces, ambiguity occurs when a class implements several interfaces with the same default methods and method signatures.

```java
public interface Alarm {
    default String turnAlarmOn() {
        return "Turning the alarm on.";
    }
    default String turnAlarmOff() {
        return "Turning the alarm off.";
    }
}
public class Car implements Vehicle, Alarm { ... }
```

The code above does not compile, since there is a conflict caused by multiple interface inheritance. To solve this ambiguity, an explicit implementation for the methods must be provided. It's even possible to use both sets of default methods:

```java
@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn() + " " + Alarm.super.turnAlarmOn();
}
@Override
public String turnAlarmOff() {
    return Vehicle.super.turnAlarmOff() + " " + Alarm.super.turnAlarmOff();
}
```

10 Autoboxing & Auto-unboxing

Boxing and unboxing is when primitive types are converted to their wrapper class equivalents, or vice versa respectively. For example:

```
int x = 3;

Integer i = x; // Autoboxing -> compiler replaces the line as such:
Integer i = Integer.valueOf(x);

int y = i; // Auto-unboxing -> compiler replaces the line as such:
int y = i.intValue();
```

Note: `Integer.valueOf()` **factory method** returns the same (cached) object for values [-128, 127]

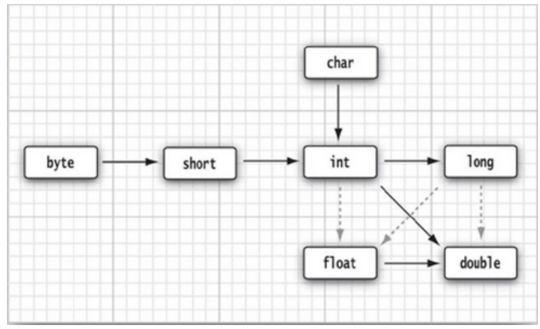
```
int a1 = 1000, a2 = 1000;
a1 == a2; // true, no auto-boxing and semantic equality is checked

Integer c1 = 100; int c2 = 100;
c1 == c2; // true , c1 is auto-unboxed and semantic equality is checked

Integer b1 = 1000, b2 = 1000;
b1 == b2; // false , no auto-unboxing and referential equality is checked

Integer d1 = 100, d2 = 100;
d1 == d2; // true, no auto-unboxing and referential equality is checked
```

- Cached instances exist for the wrapper classes `Boolean, Byte, Short, Integer, Long, Character`.
- `Character` caches values in range of [0, 127].
- The `Byte, Short, Integer,` and `Long` cache values in range [-127, 128].
- No cached instances exist for `Float` and `Double`

10.1 Typecasting and Type Conversion

A widening **(primitive)** conversion does not lose information about the overall magnitude of a numeric value.

A narrowing **(primitive)** conversion may lose information about the overall magnitude of a numeric value and may also lose precision and range. An error may be thrown if explicit typecasting is not performed:




```
Error: incompatible types: possible lossy conversion from double to float
```

A widening **(reference)** conversion is that from S to T, provided `S <: T`. The compiler can prove its correctness and therefore it never throws an exception at run time.

An example of narrowing **(reference)** conversion is that from S to T, provided that `S :> T`. This requires **explicit typecasting**, which may throw `ClassCastException` at runtime if the cast type is invalid.

11 Generics

Generic methods allow a set of related methods to be specified with a single method declaration. Generic classes allow a set of related types to be specified with a single class declaration.

Generics also provide type safety as it allows invalid types to be caught at compile time.

Note: For example, `Pair<T>` is a generic class. `Pair<Integer>` is a parameterised class, where `T` is the formal type parameter and `Integer` is the type argument.

11.1 Limitations

1. A variable of a generic type cannot be set as a static field for a class:

```
static T y;

| Error:
| non-static type variable T cannot be referenced from a static context
```

But we can return a generic type result from a static method:

```
static <T> T foo(T t) { return t; } // compiles
static T foo (T t) { return t; } // does not compile
```

2. Primitives are not allowed as a generic type:

```
Pair<int> x = new Pair<>(); // does not compile
Pair<Integer> x = new Pair<>(); // compiles
```

Generics allow classes and methods that use any reference type to be defined without resorting to using the `Object` type. It enforces type safety by binding the generic type to a specific given type argument at compile time. Attempts to pass in an incompatible type would lead to compilation error.

The diamond operator <> helps the compiler to verify type safety, for instance, that `List<A>` holds objects of type `A`. All generic information will be replaced with concrete types after compilation due to type erasure.

11.2 Bounded and Unbounded Generics

Bounded type parameters are used to place constraints on type arguments in a parameterized type. For example, a method that operates on numbers may only want to accept instances of `Number` or its subclasses.

To declare a bounded type parameter:

```
<T extends superClassName>

<T extends Interface>

<T extends superClassName & Interface1 & Interface2>
```

11.3 Wildcard

A wildcard is simply the symbol `?` which stands for an unknown type. Consider the following code:

```
void print(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

This code looks like it would work for any `Collection`, except that it doesn't. Only an actual `Collection<Object>` will be able to be passed in, while any other `Collection` will throw an error.

We can fix this by writing the function as:

```
void print(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

And we can now call it with any type of collection. `Collection<?>` is a collection whose element type matches anything.

Notice that within the function, we can still read elements from `c` and give them type `Object`. This is always safe, since whatever the actual type of the collection, it does contain objects.

However, if we declare a new object with type `?`, we cannot just add anything into it.

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // does not compile
c.add(1); // does not compile
c.add("Hello"); // does not compile
c.add(null); // compiles
```

Since we don't know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection.

Therefore, any parameter we pass to add would need to be a subtype of ?. Since the type is unknown, we cannot pass anything in. The sole exception is null, which is a member of every type. (This is similar to <? extends Object>)

On the other hand, given a List<?>, we can call get() and make use of the result. The result type is an unknown type, but we always know that it has to be a subtype of Object. It is therefore safe to assign the result to a variable of type Object or pass it as a parameter where the type Object is expected.

11.4 Bounded and Unbounded Wildcards

11.4.1 Upper Bounded (Extends)

The wildcard declaration of List<? extends Number> crudely means List<anything that is-a Number>

```
List<? extends Number> foo = new ArrayList<Number>();  // Number <: Number
List<? extends Number> foo = new ArrayList<Integer>(); // Integer <: Number
List<? extends Number> foo = new ArrayList<Double>();  // Double <: Number
```

Given that foo is of type List<? extends Number>, what type of objects are you guaranteed to read from it:

- Number is allowed because any of the lists that could be assigned to foo contain a Number or a subclass of Number
- Integer is not allowed because foo could be pointing at a List<Double>.
- Double is not allowed because foo could be pointing at a List<Integer>.

What type of object could you add to foo:

- Integer is not allowed because foo could be pointing at a List<Double>.
- Double is not allowed because foo could be pointing at a List<Integer>.
- Number is not allowed because foo could be pointing at a List<Integer>.

In fact, no object can be added to List<? extends T> because it is not guaranteed what kind of List foo is pointing to, meaning that there is no guarantee that some object is allowed in that List. The only guarantee is that foo can be read from, and that will yield an object of type T or subclass of T.

11.4.2 Lower Bounded (Super)

The wildcard declaration of `List<? super Integer>` crudely means `List<anything that Integer is>`

```
List<? super Integer> foo = new ArrayList<Integer>();  // Integer :> Integer
List<? super Integer> foo = new ArrayList<Number>();   // Number :> Integer
List<? super Integer> foo = new ArrayList<Object>();   // Object :> Integer
```

Given that `foo` is of type `List<? super Integer>`, what type of objects are you guaranteed to read from it:

- `Number` is not allowed because `foo` could be pointing to `List<Object>`.
- `Integer` is not allowed because `foo` could be pointing at a `List<Number>`.
- `Object` is allowed.

What type of object could you add to `foo`:

- `Integer` and subclasses of `Integer` are allowed
- `Double` is not allowed because `foo` could be pointing at a `List<Integer>`.
- `Number` is not allowed because `foo` could be pointing at a `List<Integer>`.
- `Object` is not allowed because `foo` could be pointing at a `List<Integer>`.

There are two ways to write a function that takes in a generic:

```
void printNumbers(List<? extends Number> list){
    for (Number i : list) {...}
}
<T extends Number> void printNumbers(List<T> list){
    for (Number i : list) {...}
}
```

11.5 PECS

Producer Extends - If the function uses a `List` to produce values (i.e. it reads objects of type `T` from the list), then `List<? extends T>` should be used. But this list cannot be added to.

Consumer Super - If the function uses a `List` to consume values (it writes objects of type `T` into the list), then `List<? super T>` should be used. But only objects of type `Object` can be read from this list.

If the function needs to both read from and write to a list, then no wildcards should be used, e.g. `List<Integer>`.

Example (Notice that `src` is **producing** and `dest` is **consuming**)

```
public class Collections {
    public static <T> void copy(List<? extends T> src, List<? super T> dest) {
        for (int i = 0; i < src.size(); i++) { ... }
    }
}
```

12 Type Erasure

To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.
- Ensure no new classes are created for parameterised types; consequently, generics incur no runtime overhead.

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or `Object` if the type parameter is unbounded.

Consider the following example:

```java
public class Node<T> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
    // more code...
}
```

Because the type parameter `T` is unbounded, the Java compiler replaces it with `Object`:

```java
public class Node {
    private Object data;
    private Node next;
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Object getData() { return data; }
    // more code...
}
```

In the following example, the generic `Node` class uses a bounded type parameter:

```java
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
    // more code...
}
```

Therefore, the compiler replaces the bounded type parameter `T` with the first bound class, `Comparable`:

```java
public class Node {
    private Comparable data;
    private Node next;
    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Comparable getData() { return data; }
    // more code...
}
```

The compiler also erases type parameters in generic method arguments. Consider the following:

```java
// Counts the number of occurrences of elem in arr.
public static <T> int count(T[] arr, T elem) {
    int count = 0;
    for (T e : arr)
        if (e.equals(elem))
            count++;
    return count;
}
```

Because `T` is unbounded, the Java compiler replaces it with `Object`:

```java
public static int count(Object[] arr, Object elem) {
    int count = 0;
    for (Object e : arr)
        if (e.equals(elem))
            count++;
    return count;
}
```

For bounded type parameters in generic method arguments:

```
class Shape { ... }
class Circle extends Shape { ... }
class Rectangle extends Shape { ... }
public static <T extends Shape> void draw(T shape) { ... }
```

The Java compiler replaces `T` with the first bound class, `Shape`:

```
public static void draw(Shape shape) { ... }
```

Implications of Type Erasure

Java does not allow to have two methods with signatures that are differentiated by generics:

```
class A {
    void foo(List<Integer> integerList) {}
    void foo(List<String> stringList) {}
}
```

```
|  Error:
|  name clash: foo(java.util.List<java.lang.String>) and
|  foo(java.util.List<java.lang.Integer>) have the same erasure
```

13 Exceptions and Error Handling

An **unchecked** exception is an exception caused by a programmer's errors. They should not happen if perfect code is written. `IllegalArgumentException`, `NullPointerException`, `ClassCastException` are examples of unchecked exceptions. Generally, unchecked exceptions are not explicitly caught or thrown. They indicate that something is wrong with the program and cause run-time errors.

A **checked** exception is an exception that a programmer has no control over. Even if the code written is perfect, such an exception might still happen. The programmer should thus actively anticipate the exception and handle them. For instance, when we open a file, we should anticipate that in some cases, the file cannot be opened. `FileNotFoundException` and `InputMismatchException` are two examples of **checked** exceptions. A checked exception **must be handled (i.e. checked)**, or else the program will not compile.

In Java, **unchecked exceptions are subclasses** of the class `RuntimeException`.

The caller of the method that **generates** (i.e. `throw new` and `throws`) an exception **need not catch** the exception. The caller can pass the exception to its caller, and so on if it is not the right place to handle it.

An exception, if not caught, will propagate automatically down the stack until either, it is caught or if it is not caught at all, resulting in an error message displayed to the user.

For instance, the following toy program would result in `IllegalArgumentException` being thrown out of main and displayed to the user.

```
class NegativeRadiusError {
  static Circle createCircles() {
    int radius = 10;
    for (int i = 0; i <= 10; i++) {
        new Circle(new Point(1, 1), radius--);
    }
  }
  public static void main(String[] args) {
    createCircles();
  }
}
```

This program won't compile because the checked exception `FileNotFoundException` is not handled.

```
// Assume FileReader constructor throws FileNotFoundException
class ExceptionDemo {
  static FileReader openFile(String filename) {
    return new FileReader(filename);
  }
  public static void main(String[] args) {
    openFile();
  }
}
```

```java
// Option 1: invoking method handles exception directly
class ExceptionDemo {
  static FileReader openFile(String filename) {
    try {
      return new FileReader(filename);
    } catch (FileNotFoundException e) {
      System.err.println("Unable to open " + filename + " " + e);
    }
  }
  public static void main(String[] args) {
    openFile();
  }
}
```

Alternatively, `openFile()` can pass the exception to the caller instead of catching it.

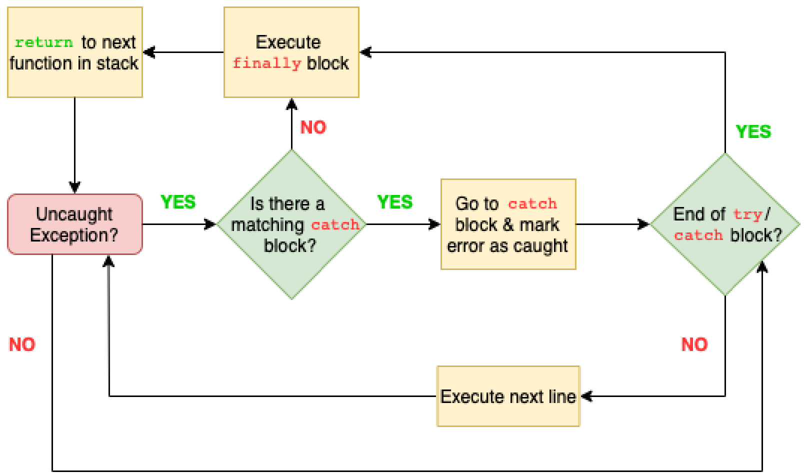
```java
// Option 2: invoking method passes exception to caller
class ExceptionDemo {
  static FileReader openFile(String filename) throws FileNotFoundException {
    return new FileReader(filename);
  }
  public static void main(String[] args) {
    try {
      openFile();
    } catch (FileNotFoundException e) {
      // warn user and pop up dialog box to select another file.
    }
  }
}
```

What should not happen is the following:

```java
// Option 3 (BAD): Exception passed to user
class ExceptionDemo {
  static FileReader openFile(String filename) throws FileNotFoundException {
    return new FileReader(filename);
  }
  public static void main(String[] args) throws FileNotFoundException {
    openFile();
  }
}
```

In the code above, neither takes the responsibility to handle it and the user ends up with the exception.

13.1 Control Flow for Exception Handling



13.2 Overriding Method that Throws Exceptions

When overriding a method that throws a checked exception, the overriding method must throw only the same, or a more specific checked exception, than the overridden method. This follows from **Liskov Substitution Principle**. The caller of the overridden method cannot expect any new checked exception beyond what has already been "promised" in the method specification.

13.3 Do's and Don'ts

13.3.1 Do Catch Exceptions to Clean Up

Consider the example earlier, where `m1()`, `m2()` and `m3()` do not handle the checked exception `E2`. Also, suppose that `m2()` allocated some system resources (e.g., temporary files, network connections) at the beginning of the method, and deallocates the resources at the end of the method.

By not handling the exception, the code that deallocates these resources does not get called when an exception occurs. It is better for `m2()` to catch the exception so that it can handle the resource deallocation in a `finally` block. If there is a need for the calling methods to be aware of the exception, `m2()` can always re-throw the exception:

```
public void m2() throws E2 {
  try {
    // setup resources
    m3();
  } catch (E2 e) {
    throw e;
  } finally {
    // clean up resources
  }
}
```

13.3.2 Do Not Break Abstraction Barrier

```
class ClassRoster {
  public Students[] getStudents() throws FileNotFoundException { ... }
}
```

`FileNotFoundException` being thrown leaks the fact that the information is read from a file. If the implementation to reading from an SQL database, the exception thrown may need to be changed to something else such as `SQLException`.

This would also require the caller to change their exception handling code accordingly. Therefore, it is better to handle implementation-specific exceptions within the abstraction barrier.

13.3.3 Do NOT Use Exception As a Control Flow Mechanism

Exceptions are meant to handle unexpected errors, not to handle the logic of the program:

Bad Example

```
try {
 obj.doSomething();
} catch (NullPointerException e) {
 doTheOtherThing();
}
```

Fixed Example

```
if (obj != null) {
  obj.doSomething();
} else {
  doTheOtherThing();
}
```

Not only is this less efficient, but it also might not be correct, since a `NullPointerException` can be triggered by something else other than `obj` being null.

14 Type Inference

An example of type inference is the use of <> when creating an instance of a generic type:

```java
Pair<String,Integer> p = new Pair<>();
Pair<String,Integer> p = new Pair<String,Integer>(); // equivalent
```

```java
// In the following examples, assume Circle <: Shape <: GetAreable <: Object
class A {
  // checks if obj is in array
  public static <S> boolean contains(Array<? extends S> array, S obj) { ... }
}
A.<Shape>contains(new Array<Circle>(0), shape); // explicit use of type argument/witness
A.contains(new Array<Circle>(0), shape); // equivalent due to type inference
```

The type inference process looks for all possible types that match. In this example, the type of the two parameters must match:

- An object of type `Shape` is passed to `S`. Therefore, `S` must be `Shape` or a supertype (due to widening type conversion).
- An `Array<Circle>` is passed to `Array<? extends S>`. Since widening type conversion occurs, the compiler finds all possible `S` such that `Array<Circle> <: Array<? extends S>`. This is true only if `S` is `Circle` or a supertype.
- Intersecting the two lists yields `Shape` or one of its supertypes: `GetAreable` and `Object`.
- The compiler chooses the most specific type among these: `Shape`.

14.1 Unexpected Consequences

```java
class A {
    public static <T> boolean contains(T[] array, T obj) { ... }
}
String[] strArray = new String[] { "hello", "world" };
A.<String>contains(strArray, 123); // type mismatch error
A.contains(strArray, 123); // compiles!?
```

- `strArray` has the type `String[]` and is passed to `T[]`. So `T` must be `String` or a supertype (`Object`). `Object` is possible since Java array is covariant.
- `123` is passed as type `T`. The value is treated as `Integer` and, therefore, `T` must be `Integer` or a supertype (`Object`).
- Intersecting the two lists yields only `Object`.

Therefore, the code above is equivalent to `A.<Object>contains(strArray, 123)`

14.2 Target Typing

Target typing is a case of type inferencing that involves the type of the expression also. Thus the processes are similar, but with an added constraint.

```
class A {
    public static <T extends GetAreable> T findLargest(Array<? extends T> array) { ... }
}
Shape o = A.findLargest(new Array<Circle>(0));
```

- The returning type of `T` must be a `Shape` or a subtype.
- Due to the bound of the type parameter, `T` must be `GetAreable` or a subtype
- `Array<Circle>` must be a subtype of `Array<? extends T>`, so T must be `Circle` or a supertype.
- Intersecting the three lists yields 2 possibilities: `Shape` and `Circle`.
- The most specific one is `Circle`, therefore: `Shape o = A.<Circle>findLargest(new Array<Circle>(0))`

14.3 Summary

Let `C` be some concrete type, and `S` be a generic

Replace `A` with the type argument to find the possible inferences

Type Parameter	Allow `A` if...	`S` could be any type that...
`S or S[]`	N.A	`A` is
`<? extends S>`	N.A	`A` is
`<? super S>`	N.A	is-a `A`
`<? extends C>`	`A` is-a `C`	N.A
`<? super C>`	`C` is-a `A`	N.A